University of Zurich[UZH]

Department of Informatics

Managing Temporal Graph Data While Preserving Semantics

A dissertation submitted to the Faculty of Economics, Business Administration and Information Technology of the University of Zurich

for the degree of Doctor of Science

by Jonas Tappolet
from Zurich, ZH, Switzerland

Accepted on the recommendation of
Prof. Abraham Bernstein, Ph.D.
Prof. Dr. Michael Böhlen

2012

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich herewith permits the publication of the aforementioned dissertation without expressing any opinion on the views contained therein.

Zurich, April 4, 2012

Head of the Ph.D. committee for informatics: Prof. Abraham Bernstein, Ph.D.

Abstract

This thesis investigates the introduction of time as a first-class citizen to *RDF*-based knowledge bases as used by the *Linked Data* movement. By presenting EvoOnt, a use-case scenario from the field of software comprehension we demonstrate a particular field that (1) benefits from the Semantic Web's tools and techniques, (2) has a high update rate and (3) is a candidate-dataset for *Linked Data*. EvoOnt is a set of *OWL* ontologies that cover three aspects of the software development process: A source code ontology that abstracts the elements of object-oriented code, a defect tracker ontology that models the contents of a defect database (a.k.a. bug tracker) and finally a version ontology that allows the expression of multiple versions of a source code file. In multiple experiment we demonstrate how Semantic Web tools and techniques can be used to perform common tasks known from software comprehension. Derived from this use case we show how the temporal dimension can be leveraged in *RDF* data. Firstly, we present a representation format for the annotation of *RDF* triples with temporal validity intervals. We propose a special usage of named graphs in order to encode temporal triples. Secondly, we demonstrate how such a knowledge base can be queried using a temporal syntax extension of the *SPARQL* query language.

Next, we present two indexing structures that speed up the processing and querying time of temporally annotated data. Furthermore, we demonstrate how additional knowledge can be extracted from the temporal dimension by matching patterns that contain temporal constraints. All those elements put together outlines a method that can be used to make the datasets published as *Linked Data* more robust to possible invalidations through updates of liked datasets. Additionally, processing and querying can be improved through sophisticated index structures while deriving additional information from the history of a dataset.

Zusammenfassung

Diese Arbeit betrachtet die Einführung von Zeit als eine eigene Dimension in *RDF*-basierten Wissensrepräsentationen wie sie zum Beispiel in Anwendungen wie *Linked Data* eingesetzt werden. Wir präsentieren EvoOnt als Anwendungsbeispiel aus dem Gebiet der Software Analyse um zu zeigen dass (1) die Techniken und Tools des Semantic Web einen wertvollen Beitrag zur Software Analyse liefern, (2) Software Entwicklungsdaten eine sehr hohe Aktualisierungsrate aufweisen und (3) dass diese Daten ein potenzieller Kandidat für die Publikation als *Linked Data* wären. EvoOnt ist eine Sammlung von drei Ontologien welche drei Aspekte der Software Entwicklung abdecken: Die Abstrahierung von objektorientiertem Software Quellcode, eine Ontologie welche Software-Defekte Datenbanken abdeckt (auch bekannt als Bugtracker) und schliesslich eine Ontologie welche die verschiedenen Versionen von Quellcode repräsentiert welche typischerweise in einem Versionierungssystem gespeichert sind. Mit einer Vielzahl von Experimenten zeigen wir wie Semantic Web Techniken im Gebiet der Software Analyse angewendet werden können. Basierend auf diesem Anwendungsfall zeigen wir wie die Vorteile einer zeitliche Annotation von *RDF*-Daten. Zuerst präsentieren wir ein Repräsentationsfrormat um *RDF*-Tripel

mit Gültigkeitsintervallen zu anotieren. Wir schlagen eine spezielle Nutzung von *Benannten Graphen* um temporale Tripel auszudrücken. Danach zeigen wir wie solche Wissensrepräsentationen verarbeitet und abgefragt werden können mithlife einer temporalen Erweiterung der Abfragesprache *SPARQL*. Folgend präsentieren wir zwei Index Strukturen welche welche die Verarbeitung und Abfrage von temporalen Daten beschleunigen. Zuletzt demonstrieren wir wie die zeitliche Dimension genutzt werden kann um zusätzliches Wissen aus den Daten durch das auffinden von Mustern in den Daten basierend auf temporalen Mustern. Alle obigen Elemente zusammen beschreiben eine Methode welche Anwendungen wie *Linked Data* robust machen gegenüber Veränderungen in verlinkten Datensätzen während gleichzeitig die Verarbeitungsgeschwindigkeit verbessert wird und zusätzliche Wissen extrahiert werden kann.

Acknowledgements

First and foremost, I would like to thank my advisor Professor Abraham Bernstein who introduced me to research and gave me the opportunity to conduct my research under his supervision. I am very grateful for his expertise and his ability to guide me through difficult stages of my thesis. My thanks also go to Prof. Dr. Michael Böhlen to be willing to co-advise my thesis and providing valuable input.

I was lucky to work with great people in our research group, DDIS. Numerous discussions helped gaining new perspectives on the topic, oftentimes in the context of memorable social activities.

Particularly, I would like to thank Christoph Kiefer and Ester Kaufmann who accompanied my transition from a MSc. to a Ph.D. student. Katharina Reinecke, Adrian Bachmann and Thomas Scharrenbach for their great company as fellow Ph.D. students, who were never too busy to discuss a problem and brainstorm about a possible solution. Patrick Minder, Lorenz Fischer, Cosmin Basca, Florea Serban and Philip Stutz for many discussions about different topics and concepts.

My thanks extend to my family; my parents, who made my studies in informatics possible, which was the foundation for this Ph.D.

thesis. And finally, I am very grateful to have Julia in my life, for her loving support and patience throughout my journey so far …

Contents

Part I

Synopsis

1

Introduction

The Internet is a huge pool of information, for instance, consider *Goolge*'s search index which contained in 2008 more than 1 trillion pages[1]. This vast amount of information is tailored to be read by humans through a web-browser. Although all the information on the Web is transported by machines, it does not come in a format that would make the information accessible to them. The Semantic Web [Berners-Lee and Hendler, 2001] fills this gap by making information being published on the internet readable and meaningful to non-human consumers such as automated agents. One particular branch of the Semantic Web covers the publication of datasets which link to each other in a similar way as hyperlinks known from webpages designed for humans do. This movement is called *Linked Data* [Heath and Bizer, 2011] and aims at bringing the notion of a hyperlink to datasets published *e.g.* by governments or NGOs. By following the links in the datasets a consuming software agent can jump from one dataset to another leveraging combined information.

[1] `http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html`

Linked Data is based on standards such as *RDF* graphs [Klyne and Carroll, 2004a] with no central curation authority due to its foundation as a community effort. This poses the problem of publishing updated versions of a dataset that potentially invalidate links created from other datasets or change their semantics. It is a shortcoming of *RDF* as it does not natively support the expression of multiple versions of the same knowledge base[2].

As an example, consider the US Census dataset[3]. The US Census is surveyed every 10 years which means that an additional datasets is added to the *Linked Data* cloud. Instead of adding the new dataset, the existing one can alternatively be updated / overwritten with the latest values of the newest census. Both ways of adding new information have implications on the graph elements that link to the altered dataset. Either they link to the old dataset without having the logical connection to the new dataset, and, thus, are lacking the knowledge thereof. Alternatively, (in case the new information overwrites the old one) they link to new elements that may have changed their semantics with the updates possibly rendering the connection meaningless. In the classic Web (with nodes containing human-readable content, *i.e.* web pages) it is immediately obvious to a reader that a page contains a broken link or does not contain the expected content. Human understanding helps resolving broken links and wrong content by using alternative ways of retrieving that information (*e.g.* by using a search engine). Creative approaches to problem solving of that kind are currently not an option for an automated agent that consumes linked data.

[2] *RDF* offers the possibility to annotate triples using reification. The YAGO2[Hoffart et al., 2010] ontology uses this mechanism (among others) to annotate temporal and spatial data to instances. Reification for temporal validity creates an at least 4-fold data overhead.

[3] `http://www.rdfabout.com/demo/census/`

Therefore, this thesis argues that *Linked Data* needs some mechanisms to add support evolving data by treating the temporal dimension as a first-class citizen in *RDF* knowledge bases. To that extent we are first presenting a use-case scenario from software comprehension which is a candidate for *Linked Data* but has a very high update rate and therefore builds the foundation of our argument for a temporal annotation of *Linked Data*.

Through the fact that in a temporal knowledge base data does not get deleted or overwritten (a deletion becomes the annotation with a temporal validity that lies in the past) such a knowledge base is doomed to be ever growing in size. Through indexing techniques this problem can be weakened, and, additionally, one can extract additional information from the history information of a dataset. To that extent this thesis investigates a representation format for temporal *RDF* together with a query language. Furthermore, we show how indexing can improve the processing time of temporal knowledge bases by proposing and evaluating two structures. Finally we demonstrate how additional knowledge can be extracted using a temporal pattern language which is based on timed automata.

1.1 Hypotheses

From the above mentioned status-quo and its current shortcomings, we derive a set of hypotheses that lay the foundation of this thesis. We will then show the connection between the hypotheses and the contribution of this thesis.

HYPOTHESIS 1 Software comprehension can benefit from the publication in the *Linked Data* and the techniques that are offered by the Semantic Web

Based on the inherent premises of the Semantic Web—a graph-based approach with world-wide unique identifiers with a logic-based foundation—Hypothesis 1 assumes that the application of these principles to the field of software comprehension can be beneficial in terms of reduction of custom-written code for analyses and the better exchangeability of extracted meta-information about the software engineering process. Furthermore, we assume that the publication of software comprehension data (*e.g.* source code information) as *Linked Data* can open new analysis possibilities by enabling links between multiple versions of different software projects hosted on various locations all over the world. The fact that multiple versions are stored can lead to the problem that a huge amount of overhead is generated (through those elements that did not change between two versions). Therefore, we argue that a more sophisticated storage format is needed.

HYPOTHESIS 2 Temporal *RDF* knowledge bases are superior to their versioned counterparts because they represent the data more efficiently in terms of procesing time.

As described above, we hypothesize that a temporal knowledge base is more efficient in the processing of data compared to the versioned counterpart as it does not store redundancy (*i.e.*, the elements that did not change between versions). We implicitly assume that an accessible temporal dimensions allows optimization steps when organizing, storing and retrieving the information from a temporal *RDF* knowledge base. This is possible because the temporal dimen-

sion is represented in a standardized way and does not depend on design decisions of a knowledge engineer.

HYPOTHESIS 3 Temporal *RDF* knowledge bases are able to derive new knowledge from the stated fact by using the semantics of the temporal dimension.

Having the temporal dimension accessible, Hypothesis 3 assumes that this information can be used to derive additional information. In a Semantic Web environment the term "deriving new information" is tightly coupled with the term reasoning (to be more precise: inferencing). As standard reasoning in the semantic web covers the application of rules to a set of stated facts, the research question is: Can we derive new information by applying *temporal* rules to a set of stated temporal *RDF* information?

In the next section we are going to connect the above hypotheses to the conducted work and the contributions of this thesis.

1.2 Contribution

In this section we are going to present the contributions of this thesis. We will succinctly evaluate each contribution against our hypotheses which were presented in the previous section. In summary, the contribution of this thesis is as follows:

- EvoOnt, a Semantic Web based software comprehension framework which is also a use-case scenario for a dataset that has a high temporal fluctuancy
- An efficient format for temporal annotation of *RDF*.

- A query syntax based on *SPARQL* (*τ-SPARQL*) for the representation of the temporal annotations within the query.

- An index structure that entails the interval relations which can be queried with *τ-SPARQL*.

- The definition of a temporal pattern language and its mapping on timed automata to entail new knowledge.

Figure 1.1 gives a graphical overview of the different parts of this thesis and their respective categories of contribution.

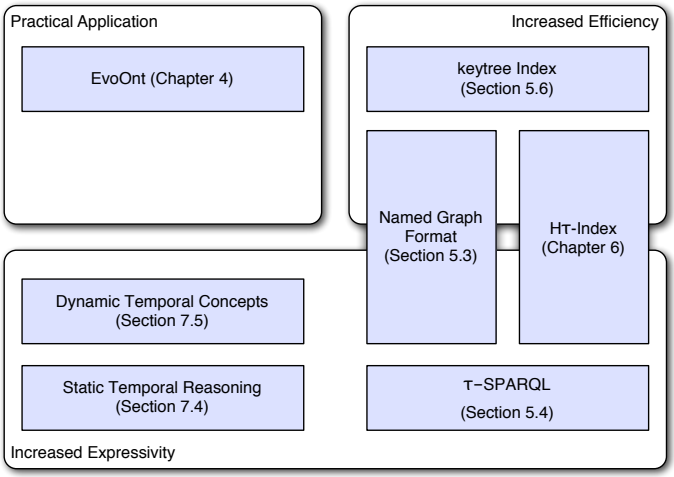


Figure 1.1: Overview of the different parts of this thesis

In the following, we will further discuss the conducted work towards the contributions of this thesis.

1.2.1 Semantic Web Enabled Software Analysis

In Chapter 4 of Part II we are employing Semantic Web techniques to the field of software analysis. There, we outline a possible *Linked Data* scenario where different (open-source) projects expose their source code to the internet through a standardized *RDF* format. This enables the automated verification of software source-code links such as library calls among various different software projects and versions.

Having a globally interlinked software source graph, new analysis possibilities arise: it would be possible to analyze whether a bug in a software project may be affecting the functionality of a second project. So far, this was only possible with offline processing of the respective source code fragments which first had to be extracted. The Ph.D. thesis of Yonggang Zhang [Zhang, 2007] gives an excellent overview over existing software comprehension frameworks and describes a similar approach to EvoOnt's software ontology.

Concluding from Chapter 4 in Part II we can support Hypothesis 1 based on the above mentioned findings. Indeed, it would be beneficial to publish software source code repositories as *Linked Data*. One finding of Chapter 4 in Part II is the slow processing of the *RDF* datasets because a versioned approach was used. In a subsequent publication (Chapter 5 of Part II) we are addressing this issue. The consolidated contribution is presented in the next section.

1.2.2 Temporal RDF Serialization Format

As mentioned in the previous section, the storage of multiple versions of the same software source-code has severe implications on the processing time. As a reason we identified the fact that software

has a change rate between versions of below five percent[4]. To that end this thesis proposes a novel format for temporal representation of *RDF* data. It uses one named graph for every time interval and stores the triples whose temporal validity belongs to this interval within the named graph. This allows the temporal annotation of each triple with a minimum of overhead (as compared to *e.g.* reification). Using standard Semantic Web techniques, the querying of the temporally annotated knowledge bases can be done within the expressivity of *SPARQL*, the standardized query language of the Semantic Web.

1.2.3 Querying of temporally annotated RDF

Using the representation format as described above, temporal constraints in queries can be mapped to the selection of the appropriate named graphs holding the triples with the corresponding properties. Since there exists a well-defined format of the named graphs and their properties, we propose a syntax that facilitates the intuitive query writing. This syntax can be mapped to standard *SPARQL* to be processed by a off-the-shelf query engine. By having a standardized way of accessing temporal information, one can use optimized access methods in order to speed up the query processing by using sophisticated indexing structures.

1.2.4 Indexing Temporal RDF

Another contribution of this thesis is the application of different indexing strategies during the evaluation of a query. We demonstrate in Chapter 5 in Part II how the query execution time can be im-

[4] we observed higher change rates but those were exceptional; you can find a plot of the change rates in Figure 5.1(b)

proved when using a temporal indexing structure which optimizes time-point queries. Using such an index structure, we were able to show that the proposed temporal *RDF* approach is superior in terms of processing time for all tested queries and datasets to, *e.g.*, storing multiple versions. Based on the findings from Chapter 5 in Part II we conclude that Hypothesis 2 is accepted as we could demonstrate a clear performance improvement when using a temporal instead of a versioned approach.

Another major contribution of this thesis is the definition of an index structure that allows the storage of elements relative to each other in a way that the Allen interval relations [Allen, 1983] can be retrieved with minimal effort in terms of processing time. To that end, Chapter 6 in Part II projects a time interval into the 2-dimensional space where the Allen interval relations can be identified as rectangular regions. For efficient storage using a B+Tree, the 2-dimensional space is reduced back to a single dimension using the Hilbert space-filling curve. The proposed algorithm for the optimized retrieval of rectangles from the Hilbert curve is demonstrated to be better performing than competing approaches. With a built-in cost function, the algorithm can adapt to different system setups with different costs for *e.g.* disk accesses and in memory processing. While the Allen intervals form one basic extraction of additional knowledge, there is more knowledge buried in the temporal dimensions which is covered in the next section.

1.2.5 Temporal Pattern Matching

The final contribution of this thesis demonstrates the matching of patterns on temporal knowledge bases. By defining a set of base patterns which can be parametrized with various attributes (cf. Chapter 7 in Part II) and providing a serialization syntax, we demonstrate the

entailment of additional knowledge from the temporal dimension using timed automata. By writing the matched elements back to the knowledge base as *RDF* triples, they can contribute to additional pattern-matches in the spirit of Semantic Web inferencing. We also demonstrate that the processing can be done in linear time thanks to the usage of timed automata. Both, the 2-dimesional indexing approach as well as the automaton-based pattern matching demonstrate the possibility to extract additional knowledge from the temporal dimension. Thus, Hypothesis 3 is considered to be accepted.

2

Limitations and Future Work

In this chapter we are going to name some limitations of the presented approaches for each partial contribution of this thesis. Furthermore, we will sketch some directions for future work based on the limitations, but also new opportunities which opened up during the investigation of the various aspects in this thesis.

2.1 Limitations

SEMANTIC SOFTWARE ENGINEERING. Our approach proposed in Chapter 4 in Part II only covers one aspect of the software engineering process. Namely the development of source code reflected by the source code itself, the reported defects and the development iterations maintained by the version control system. Take the waterfall development model [Royce, 1970] as an example[1]: Software design is only one of six major steps that constitute the software development process. In order to semantically support the whole software development process it would need an integrated approach

[1] although the waterfall model may be outdated, it still give a good overview over the different tasks in software development

that covers requirements specification, integration, testing, deployment and maintenance. There exists work that covers other parts of the software development process [Witte et al., 2007, Bossche et al., 2007]. To tap the true potential of Semantic Web enabled software development, one would need to build a semantic layer that covers the whole process [Li and Zhang, 2011, Tappolet, 2008].

REPRESENTATION FORMAT. The proposed representation format uses named graphs to annotate triples with temporal validity. This approach can become problematic when every triple is annotated with a unique time interval. The system has to cope with as many time intervals as there are triples. Although feasible, the most gain in performance is achieved when multiple triples share the same temporal validity. This problem becomes weakened when applying sophisticated indexing structures making the management of vast amount of time intervals possible.

QUERY SYNTAX. A shortcoming of our temporal *SPARQL* query language is the fact that it only extends the syntax of the query language. It was not intended (although desirable) to provide new operators for *e.g.* the matching of sequences. This would require a sophisticated query engine that supports those operators, our intent was to demonstrate the possibility to extend off-the-shelf technology in order facilitate temporal queries. Subsequent publications have covered the aspects of the querying of generic annotated knowledge bases [Lopes et al., 2010] or sequence detection in streams [Anicic et al., 2011].

TEMPORAL INDEXING. The first of the two proposed index structures, the keyTree index is currently only an in-memory index struc-

ture. Similar index structures which have on-disk capabilities (*e.g.* the time index by [Elmasri et al., 1990]) do not have the possibility to adapt the key-indices positions to the very nature of the knowledge base's spread of the temporal dimension. In those approaches, the index positions that hold the whole information about the temporal validity of a time instant are fixed (given by the disk page of the storage system).

In contrast, the second index structure proposed by this thesis, the Hτ-index is a true on-disk index structure. Similar to other multidimensional index structures (*e.g.* the UB-Tree [Bayer and Markl, 1998]) it has the downside that it assigns index positions according to the position in the 2-dimensional space not taking into account whether there are vast unused positions. In other words: the index structure does not take into account whether the index load is 100% (every possible time interval indexed) or 0% (empty). An extended version of the Hτ-index should be equipped with a side structure that gives some insight whether a requested area contains elements or not. This will prevent our algorithm from making disk accesses to empty index areas.

Temporal Pattern Matching. The proposed reasoning approach to extract temporal patterns from a *RDF* knowledge base bears some limitations. Through the usage of timed automata, the time complexity for the evaluation of the automata is linear allowing a soft-realtime processing (assuming that the triples are ordered by the value of the beginning of their validity). Nevertheless, the space complexity is potentially high (depending on the pattern) an can lead to the reaching of physical memory limits of a processing system. In order to control the space consumption, the introduction of a time-window (as used by *e.g.* c-*SPARQL* [Barbieri et al., 2010]) could prove useful. Alternatively, other memory management strategies

known from stream processing could be employed such as LRU (last recently used) and FiFo (first in first out).

2.2 Future Work

In this section we are going to outline possible directions for future extensions of the presented approach.

Semantic Software Engineering. As mentioned in the limitations section above, a logical next step for the EvoOnt approach would be to complete the sketched system in order to be able to serve as a production-grade software development support system. Mathew Perry described a spatio-temporal and thematic system in his Ph.D. thesis [Perry, 2008] which is a generic approach based on the *Oracle* database server. A more software development centric system would implement *e.g.* the protocol used in *Subversion* enabling the EvoOnt system to be connected directly to an IDE[2]. Once a developer commits a file containing source code, the connected EvoOnt system could automatically convert the source code into the representation proposed in Chapter 4 in Part II. The semantic version control system could expose a *SPARQL* endpoint that allows other version systems of this kind to run queries against the maintained code base. Whenever a developer tries to check out a piece of code, the EvoOnt representation would be transformed back into a source code file. Every change could be maintained by a temporal *RDF* knowledge base as it is described in Chaper 5 in Part II. Patterns that could be defined similar to the proposed syntax in Chapter 7 in Part II would allow to quickly trigger actions whenever certain events occur in the system. Consider the fact that a developer checks in the

[2]Integrated Development Environment, *e.g.* Eclipse, `http://www.eclipse.org`

same file many times within a short period of time. Probably the software project's manager would like to receive a notification in order to take action and review the code due to this unusual change pattern.

Temporal Index Structures. Based on the work done with the Hτ-Index, a future extension could be the side structure that tracks the index areas' load factors. The fact that the Hilbert curve is based on a four quadrant pattern can be exploited by starting at the topmost level giving the binary information which of the four quadrants has elements inside. This can be repeated for the subsequent levels. One has to decide where the trade-off lies between overhead and utility of this side structure. At some point it is probably better to access the index and risk the hitting of an empty section than keeping a very fine-grained load structure.

Another interesting extension of the Hτ-Index would be the extension from two dimensions to n dimensions. It would be worth investigating whether an extension of additional two or three dimensions could transform the Hτ-Index into a spatio-temporal index structure. Following this strain, one could investigate whether a similar connection exists between space and RCC-relations [Randell et al., 1992] was presented in this thesis between space and Allen relations. One could even further extend the index with three additional dimensions covering the subject, predicate and object of the stored triples. It would be interesting to investigate how the algorithm presented in Chapter 6 in Part II performs as there are certain side-effects to consider such as the curse of dimensionality [Bellman, 1961].

Temporal Pattern Matching. Finally, the extraction of additional knowledge from temporal *RDF* offers some additional fields for research. So far, we have considered temporal knowledge bases as

a collection of known triples with temporal annotations of validity. Processing was done by a system which had access to the full knowledge base. When changing the perspective, one could "fix" the system's view on the data and move the knowledge base through the processing system according to the temporal order. By taking this view, one ends up at a stream processing perspective which is its own field of research. Nevertheless, little research has been conducted on triple streams, which are different from number streams (*e.g.* as produced by the stock exchange) and are finer-grained as tuple streams which poses new challenges with respect to memory processing.

3

Conclusions

In this thesis we have presented an approach that enables the temporal annotation of *RDF* knowledge bases. This is of special interest when considering datasets from *Linked Data* which undergo frequent updates. In order to prove the need for such an extension, we presented an application in the field of software comprehension where Semantic Web techniques can contribute to the integration and analysis of heterogenous data sources such as source code and version control systems. We showed that in order to manage multi-versioned data, the standard methods provided by the Semantic Web have their shortcomings. In order to address this issue we presented a temporal *RDF* approach which can completely remove the redundancy of a versioned approach by introducing a minimum of overhead. With the temporal dimension being available as a first-class citizen in knowledge bases, we demonstrated that processing can be even further optimized by using sophisticated temporal indexing structures.

Additionally, we presented two methods to extract additional information from the temporal dimension. Trough a 2-dimensional index structure the time intervals and their Allen relations can be di-

rectly identified as a rectangle in space. By reducing from two to one dimension using a Hilbert space-filling curve, the index structure can be stored in a off-the-shelf B+Tree for optimized disk access. To prevent to many non-contiguous disk accesses, we presented a novel algorithm that enables the retrieval of rectangles from the Hilbert curve applying a cost function that weights the number of disk access versus the amount of overhead retrieved from the index.

A second method to extract additional information is the detection of temporal patterns. We presented four base patterns that can be parametrized and combined to form complex structures describing temporal occurrences of certain triples. By mapping those basic patterns to timed automata, the matching can be done in linear time.

Concluding, we have presented a set of methods which could be implemented as a system that allows the *Linked Data* to be extended with temporal information. Having such an extension, it is a contribution towards a long-term oriented securing of data integrity in a distributed, decentrally curated data conglomerate as *Linked Data* is intended to be.

Part II

The Publications That Constitute This Thesis

4

Semantic Web Enabled Software Analysis

Jonas Tappolet, Christoph Kiefer and Abraham Bernstein

Published in the *Journal of Web Semantics, Vol 8, Issue 2–3, 2010*

ABSTRACT One of the most important decisions researchers face when analyzing software systems is the choice of a proper data analysis/exchange format. In this paper, we present EvoOnt, a set of software ontologies and data exchange formats based on OWL. EvoOnt models software design, release history information, and bug-tracking meta-data. Since OWL describes the semantics of the data, EvoOnt (1) is easily extendible, (2) can be processed with many existing tools, and (3) allows to derive assertions through its inherent Description Logic reasoning capabilities. The contribution of

this paper is that it introduces a novel software evolution ontology that vastly simplifies typical software evolution analysis tasks. In detail, we show the usefulness of EvoOnt by repeating selected software evolution and analysis experiments from the 2004-2007 Mining Software Repositories Workshops (MSR). We demonstrate that if the data used for analysis were available in EvoOnt then the analyses in 75% of the papers at MSR could be reduced to one or at most two simple queries within off-the-shelf SPARQL tools. In addition, we present how the inherent capabilities of the Semantic Web have the potential of enabling new tasks that have not yet been addressed by software evolution researchers, e.g., due to the complexities of the data integration.

4.1 Introduction

Ever since software is being developed, there was a need to understand how code works and why developers made certain decisions. One reason is the fluctuation rate in development teams requiring new employees to familiarize with the existing code and its peculiarities. Secondly, many programmers agree that they tend to forget about a certain piece of code's structure and its rationale behind after a short period of not touching that specific component or class. Another obstacle for comprehending source code is outdated comments [Lethbridge et al., 2003] or the complete lack thereof. The problem gets aggravated when the history of source code is considered as well; since multiple versions add another dimension of complexity.

Imagine a software developer who newly joins a software engineering team in a company: He will most probably be overwhelmed by the vast amount of source code, versions, releases, and bug re-

ports floating around. In the last decades a reasonable amount of different code comprehension frameworks were proposed and implemented. These frameworks aim at facilitating the navigation through the code and the identification of certain anomalies (e.g. code smells [Fowler, 1999], anti-patterns [Lanza and Marinescu, 2006]) or the structure of the code in general (e.g. intensity of couplings between components). Most of these frameworks first convert the source code into an internal representation that serves as a basis for fast query answering. We will review a selection of such frameworks in Section 6.2.

In addition, due to the complexity of software products and the growing popularity of open source software components, modern software has become a fine-grained composition of a multitude of different libraries. A typical software product uses external libraries, for instance, for the user interface (e.g. SWT[1]), data layer abstraction (e.g. Hibernate[2]) or logging (e.g. log4j[3]). Each of those libraries in turn make use of sub-libraries that are again maintained by their own project teams. This view turns a software project that seems to be developed locally into a node in a world-embracing network of interlinked software source code or, more technically speaking, a global call-graph. Problems in software projects often happen at this edge between the project's source code and an imported library (as witnessed by the special bug category "3rd party" in bug-trackers): A bug in a library may influence the behavior of the calling component or the wrong usage of a component may lead to instabilities in the code. Therefore, this global cloud of software source code and its related information (versions, releases and bug reports) implies additional requirements for a comprehension framework. Instead

[1] `http://www.eclipse.org/swt/`
[2] `http://www.hibernate.org/`
[3] `http://logging.apache.org/log4j/index.html`

of an insular system with internal representations, each software project participating in the cloud needs to exhibit its information in an open, accessible and uniquely identifiable way. To this end, we propose the usage of semantic technologies such as OWL, RDF and SPARQL as a software comprehension framework with the abilities to be interlinked with other projects. We introduce a set of ontologies and techniques to represent software project information semantically and focus on the general abilities of semantic technologies to cover everyday problems in software comprehension (also referred to as Software Analysis). We show that semantic technologies indeed bear the potential to serve as a general-purpose framework and we believe that implicit abilities such as the strong web-based foundation including unique resource identifiers or distributed querying are the key towards a worldwide connection of different software projects.

Specifically, we present our software evolution ontology EvoOnt, which in fact is a graph-based, self-describing representation for source code and software process data, allows the convenient integration, querying, and reasoning of the software knowledge base. Together with some standard Semantic Web tools and our domain-independent iSPARQL as well as SPARQL-ML query engines, EvoOnt can help to resolve various software analysis tasks (including some in cross-project settings).
EvoOnt is a set of software ontologies based on OWL. It provides the means to store all elements necessary for software analyses including the software design itself as well as its release and bug-tracking information. Given the proliferation of OWL (as a standard), a myriad of tools allow its immediate processing in terms of visualization, editing, querying, reasoning, and debugging avoiding the need to write code or use complicated command line tools. OWL enables

handling of the data based on its semantics, which allows the simple extension of the data model while maintaining the functionality of existing tools.

Furthermore, given OWL's Description Logic foundation, any Semantic Web engine allows deriving additional assertions in the code such as orphan methods (see Section 5.4), which are entailed from base facts. To highlight EvoOnts full capabilities we used it in conjunction with two of our domain-independent SPARQL extensions iSPARQL [Kiefer et al., 2007a] and SPARQL-ML [Kiefer et al., 2008]. iSPARQL extends the Semantic Web query language SPARQL with similarity joins allowing EvoOnt users to, e.g., query for similar software entities (classes, methods, fields, etc.) in an EvoOnt dataset or to compute statistical propositions about the evolution of software projects for instance (see Section 5.2). SPARQL-ML seamlessly extends SPARQL with (two) machine learning libraries allowing, e.g., to use SPARQL queries for induction of defect prediction models.

The main contribution of this paper is the introduction of EvoOnt simplifying most typical software analysis and prediction tasks within one extendible framework. In addition, the simplicity in which EvoOnt could support the software analysis indicates that future tasks might be just as easy to handle. Last but not least, the inherent capability of the Semantic Web to process distributed knowledge bases significantly simplifies analyses among many different software projects. We show that our approach allows reducing more than 75% of the evolution analysis tasks typically conducted at the ICSE Mining Software Repository Workshop (MSR) to one (sometimes two) queries and argue that some other tasks could also be performed with some simple extensions of EvoOnt and/or iSPARQL/SPARQL-ML.

The remainder of this paper is structured as follows: next, we succinctly summarize the most important related work. Section 3 presents EvoOnt itself, which is followed by brief introductions to iSPARQL and SPARQL-ML. Section 5 illustrates the simplicity of using EvoOnt. To close the paper, Section 6 presents our conclusions, the limitations of our approach, and some insight into future work. We would like to mention that this work builds upon two of our previous publications [Kiefer et al., 2007c, Kiefer et al., 2007b].

4.2 Related Work

4.2.1 Software Comprehension Frameworks

As mentioned in the section above, a number of software comprehension frameworks have been proposed in recent research. We will give a brief overview of these frameworks with a focus on the applicability to a worldwide-interweaved scenario. One representative dating from the late 80ies is RIGI [Müller and Klashinsky, 1988]. It has a strong emphasis on the recovery of the code's architectural structure. To that end, RIGI uses an internal representation that is a graph, however, mapped to a relational data model for storage. RIGI comes with a number of different components such as a GUI and analysis component. While RIGI has a strong focus on visual analysis, it is limited to the implemented analysis methods because the internal representation is not exposed and accessible for third-party tools. The same applies to tools like CIA [Chen et al., 1990] and CShape that were implemented with a finite set of analysis tasks in mind. This is a gap that is filled by the GENOA [Devanbu, 1999] framework. It also has a graph-based internal representation, but additionally offers a formal language interface that can be used by a

multitude of different frontends (analysis tools) to retrieve information. GUPRO [Ebert et al., 2002] follows a similar approach by using a graph-based query language to access the source code information. LaSSIE [Devanbu et al., 1991], presented by the same authors as GENOA, exposes a natural language interface serving as a more intuitive way of accessing the knowledge base. All of the above mentioned approaches use internal data models with local identifiers. It is non-trivial to put those software projects into relation to their libraries and dependencies. They provide non-standardized query interfaces to access their knowledge base, if at all. Finally, the OMG (Object Management Group) specified QVT [OMG, 2008] within its ADM initiative (Architecture-Driven Modernization). QVT stands for *Query / View / Transformation* and its goal is the transformation between different object-oriented code model representations. Unlike *SPARQL*, QVT uses mostly an SQL-like relational approach instead of graph patterns. The tool support is not (yet) very comprehensive.

4.2.2 Software Exchange Formats

To address the issue that each analysis framework needs to provide its own extraction tools suitable for the internal format, generic exchange formats have been proposed. Many of the above-mentioned tools define their own format primarily for persistent storage of their internal data. With GXL [Holt et al., 2000], an effort was made to exchange software graphs between TA [Holt, 1998], TGraphs (GUPRO), RPA [Feijs et al., 1998], RSF (Rigi Standard Format) [Müller and Klashinsky, 1988] and PROGRES [Schürr et al., 1999]. It extends the tree-based XML to be able to express graphs. An earlier exchange format was CDIF (CASE Data Interchange Format),

an EIA[4] standard for exchanging data between CASE (Computer Aided Software Engineering) tools. It uses flat textual representations, which makes it human-readable. An example for a comprehension framework supporting CDIF is FAMIX, the meta-model for object-oriented source code of the MOOSE[5] project; in the earlier versions of FAMIX the CDIF format was used. Later, the successor, XMI (XML Metadata Interchange) [OMG, 1998], an XML based exchange format able to express multiple different models and even graphics was used. XMI is a standard of the OMG. Both CDIF and XMI are highly sophisticated exchange formats. Since they were designed especially for the domain of CASE tools there is a good and widespread tool support. Unfortunately, tool providers tend to extend XMI with proprietary elements resulting in an erosion of the standard. Another downside is the need for transformation between a tool's internal representation and the exchange format. This can be an error-prone and expensive step. Our approach proposes the usage of one format both for internal representation and as exchange format. In addition, neither XML nor CDIF impose the rigid usage of global identifiers in a way that RDF does. This is, as mentioned above, a precondition for inter-project software comprehension and code analysis. Finally, none of the existing exchange formats expose their semantics formally. They are usually defined in a human-readable format aiming at being implemented in tools. The advantage of self-describing and exposed semantics is the fact that tools can handle the information without the need of being developed for a certain domain of application (e.g. query languages, visualization tools or machine learning tools).

[4] http://www.eia.org/
[5] http://moose.unibe.ch/

4.2.3 Semantic Web Enabled Software Engineering

Semantic Web technologies have successfully been used in recent software engineering research. For example Dietrich [Dietrich and Elgar, 2005] proposed an OWL ontology to model the domain of software design patterns [Gamma et al., 1995] to automatically generate documentation about the patterns used in a software system. With the help of this ontology, the presented pattern scanner inspects the abstract syntax trees (AST) of source code fragments to identify the patterns used in the code.
The decision as to which software design patterns to choose is a crucial step in designing a software system. Choosing a wrong (or inappropriate) architectural design probably results in high maintenance costs and poor performance and scalability. With the proposed software evolution ontology *EvoOnt* we are, in fact, able to measure the quality of software in terms of its used design patterns. This, in combination with data from version control and a bug-tracking system, enables us to perform powerful and complex software analysis tasks (see Section 4.5).

Highly related is the work of Hyland-Wood [Hyland-Wood et al., 2006], in which the authors present an OWL ontology of *Software Engineering Concepts (SECs)*. Using SEC, it is possible to enable language-neutral, relational navigation of software systems to facilitate software understanding and maintenance. The structure of SEC is very similar to the language structure of Java and includes information about classes and methods, test cases, metrics, and requirements of software systems. Information from versioning and bug-tracking systems is, however, not modeled in SEC.
In contrast to EvoOnt, SEC is not based on FAMIX [Demeyer et al., 1999] that is a programming language-independent model to represent object-oriented software source code. EvoOnt is, thus, able to

represent software projects written in many different object-oriented programming languages.

Witte *et al.* [Witte et al., 2007] presented an approach that is similar to the idea of EvoOnt. The scope of their work is not the integration of bug, version and source code information but the connection of source code with its documentation. We believe that EvoOnt could be attached to the documentation ontologies of their work to have even more information available in the knowledge base.

Both, Mäntylä [Mäntylä et al., 2003] and Shatnawi [Shatnawi and Li, 2006] carried out an investigation of *code smells* [Fowler, 1999] in object-oriented software source code. While the study of Mäntylä additionally presented a taxonomy (*i.e.*, an ontology) of smells and examined its correlations, both studies provided empirical evidence that some code smells can be linked with errors in software design.

Happel [Happel et al., 2006] presented the *KOntoR* approach that aims at storing and querying metadata about software artifacts in a central repository to foster their reuse. Furthermore, various ontologies for the description of background knowledge about the artifacts such as the programming language and licensing models are presented. Also, their work includes a number of *SPARQL* queries a developer can execute to retrieve particular software fragments which fit a specific application development need.

Finally, we would like to point out that EvoOnt shares a lot of commonalities with *Baetle*[6] which is an ontology that heavily focuses on the information kept in bug databases, and makes use of many other well-established Semantic Web ontologies, such as the Dublin Core[7], and FOAF[8]. We merged the ideas realized in baelte with our

[6] `http://code.google.com/p/baetle/`
[7] `http://dublincore.org/documents/dcq-rdf-xml/`
[8] `http://www.foaf-project.org/`

Bug Ontology Model (see Section 4.3.3). Therefore, most members of the *Baetle* community base their work on our ontology.

As a conclusion of the related work, we believe that EvoOnt will contribute to the state of the art as follows: (1) The usage of the open and well-established RDF/OWL format can decouple the analysis tool from the data export tool (so far, an analysis tool is responsible to transform the data into its own internal format). (2) Other than existing exchange formats, EvoOnt exposes its semantics, which allows standard tools to process the data using a unified query language (*SPARQL*) including extensions such as i*SPARQL* and SPARQL-ML. Additionally, unlike CDIF or XMI, EvoOnt can be extended easily by either attaching additional ontologies or by using sub-concept specialisation. Finally (3), EvoOnt imposes the usage of globally unique identifiers, which is a main requirement for inter-project analysis.

4.3 Software Ontology Models

In this section, we describe our OWL software ontology models. Figure 4.1 shows the complete set of our ontologies and their connections between each other. We created three different models which encapsulate different aspects of object-oriented software source code: the *software ontology model (som)*, the *bug ontology model (bom)*, and the *version ontology model (vom)*. These models not only reflect the design and architecture of software, but also capture information gathered over time (*i.e.*, during the whole life cycle of the project). Such metadata includes information about revisions, releases, and bug reports. We connected our ontologies to existing ones from other domains. A bug report for example can be seen as a representation of a work flow. Therefore, we used the defined concepts of Tim Berners-Lee's

work flow ontology[9]. The following list shows the external ontologies with their description and abbreviation (prefix) used in the remaining parts of this paper.

- *doap: Description of a Project* defining concepts about a project itself as well as different version control systems (e.g. `CVS-Repository`)
- *sioc: Semantically Interlinked Online Communities*. In this ontology, concepts modeling the activities of online communication are defined.
- *foaf:* The *Friend Of A Friend Ontology* is an approach of modeling social networks, i.e., persons and the connection to each other. We use the concept `Person` to reflect human interaction within the repositories.
- *wf:* Tim Berners-Lee's *work flow ontology*. In our approach, a bug report is considered a work flow. Therefore, an issue (bug report) is a `wf:Task` which can have a `wf:NonTerminalState` (still processing) or a `wf:TerminalState` (fixed / closed bug). This idea is adopted from the *baetle* project.

4.3.1 Software Ontology Model

Our software ontology model (som) is based on *FAMIX* (FAMOOS Information Exchange Model) [Demeyer et al., 1999], a programming language-independent model for representing object-oriented source code. FAMIX and other meta-models abstract OO concepts

[9] `http://www.w3.org/2005/01/wf/`

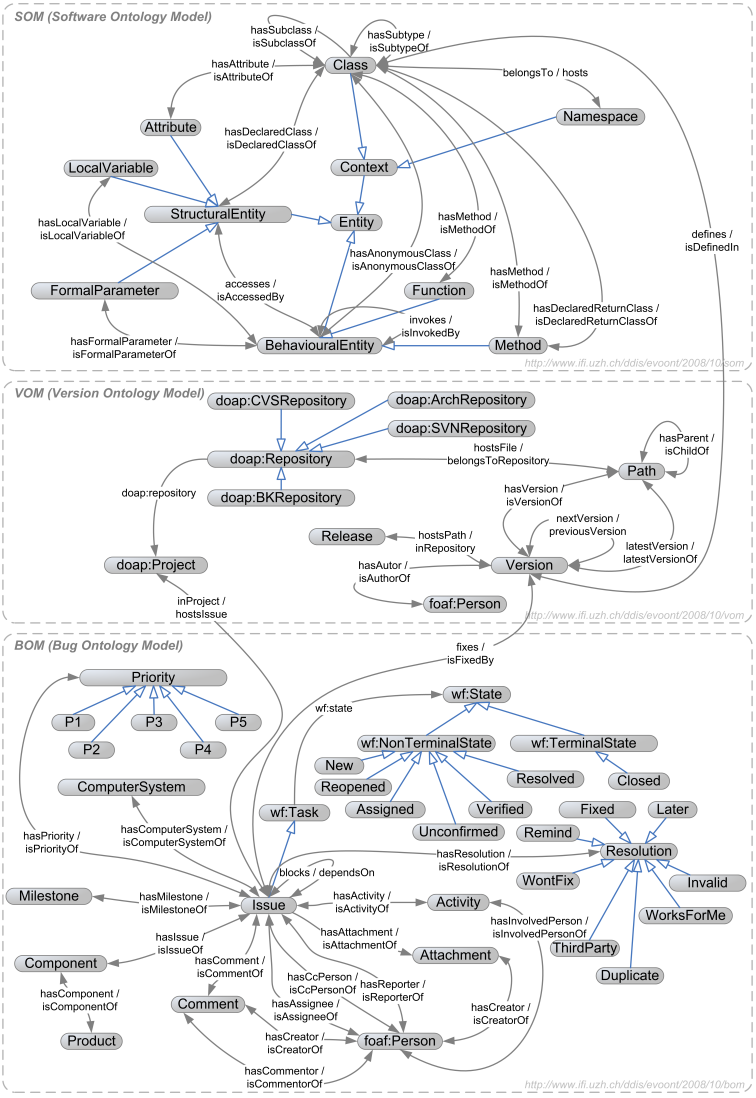


Figure 4.1: EvoOnt's three ontology models: Software (SOM), Version (VOM), and Bug (BOM) Ontology Model. Solid arrows: property relationships; hollow arrows: class inheritance.

in a similar way. Therefore, the choice of using FAMIX is not irrevocable. Other meta-models such as Lethbridge *et al.*'s DMM (Dagstuhl Middle Metamodel)[Lethbridge et al., 2004] can partially (or completely) be attached to the current ontology to, for example, express control structures such as `while`, `switch`, or `if`. This can be achieved by using `sameAs` relations or, in case of a finer-grained definition, a subclass definition — another advantage of RDF/OWL's exposed semantics.

On the top level, the ontology specifies `Entity` that is the common superclass of all other entities, such as `BehaviouralEntity` and `StructuralEntity` (see Figure 4.1 (top)). A `BehaviouralEntity` represents the definition of a behavioural abstraction in source code, *i.e.*, an abstraction that denotes an action rather than a part of the state (achieved by a method or function). A `StructuralEntity`, in contrast, represents the definition in source code of a structural entity, *i.e.*, it denotes an aspect of the state of a system [Demeyer et al., 1999] (*e.g.*, variable or parameter).

When designing our OWL ontology, we made some changes to the original FAMIX: we introduced the two new classes `Context` and `Namespace`, the first one being the superclass of the latter one. `Context` is a *container class* to model the context in which a source code entity appears. `Namespace` (not to confuse with an RDF namespace, *i.e.* URI) denotes a hierarchical identifier for source code (*e.g.*, in Java this concept is called *package*). Taking advantage of RDF's graph-based foundation, RDF/OWL now allows us to elegantly model so-called *association classes*, such as methods accessing a variable with the property `accesses` having the domain `BehaviouralEntity` and range `StructuralEntity`.

4.3.2 Version Ontology Model

The goal of our version ontology model (vom) is to specify the relations between files, releases, and revisions of software projects and the projects themselves (See Figure 4.1 (middle)). We took the data model of Subversion[10] as a blueprint for vom. To that end, we defined the three OWL classes `Path`, `Release`, and `Version` as well as the necessary properties to link these classes. A `Path` denotes a constant, non-temporal entity which could also be seen as a file, but without content or any meta-data. According to the name, a `Version` relates to a file belonging to a `Path` which is valid for a certain timespan between its predecessor and successor version. A `Version` has content and meta-data as we would expect it from a classical file in a file system (i.e., author or creation date) and, as a characteristic to a versioning system, a comment and a revision number. As container entity we import `doap:Project` and `doap:Repository` (with its subclasses) from the *Description Of A Project* ontology. Every path is connected to `Repository` using the `belongsToRepository` object property. On the other hand, a `Repository` is connected to a `Project` that it is part of using the `doap:repository` property. A `Path`, for example, has a number of revisions and, therefore, is connected to `Revision` by the `hasRevision` property. At some point in time, the developers of a software project usually decide to publish a new release, which includes all the revisions made until that point. In our model, this is reflected by the `isReleaseOf` property that relates `Release` with one or more `Version`s.

[10] `http://subversion.tigris.org/`

4.3.3 Bug Ontology Model

Our bug ontology model (bom) (see Figure 4.1 (bottom)) is inspired by the bug-tracking system *Bugzilla*.[11] `Issue` is the main class for specifying bug reports. As stated above, we consider a bug report to be a task from a work flow. Therefore, `Issue` is a subclass of `wf:Task` defined in the workflow ontology. It is connected to a `foaf:Person`, which stands for any person interacting with the bug tracking system, *i.e.*, the reporter of a bug or a developer that fixes a bug. `Activity` stands for every change made to a bug report. This can be, *e.g.*, the change of the status or resolution of a bug report or the current state of the bug.[12] `Issue` has a connection to `Version` (see Section 4.3.2) via the `isFixedBy` property. This way, information about which file version successfully resolved a particular bug can be modeled, and vice versa, which bug reports were issued for a specific source code file.

4.4 Semantic Web Query Methods for Software Analysis

The contribution of our paper is to show how software analysis tasks can be vastly simplified using EvoOnt and off-the-shelf Semantic Web tools. To ensure that this paper is self-contained we succinctly review the two non-standard, off-the-shelf, domain-independent Semantic Web query approaches used in this paper: i*SPARQL* (imprecise *SPARQL*) and SPARQL-ML (*SPARQL* Machine Learning). For details about these approaches, refer to [Prud'hommeaux and Sea-

[11] `http://www.bugzilla.org/`
[12] `https://bugs.eclipse.org/bugs` shows various concrete examples.

borne, 2008], [Kiefer et al., 2007a], and [Kiefer et al., 2008] respectively.

4.4.1 iSPARQL

iSPARQL[13] is an extension of *SPARQL* [Prud'hommeaux and Seaborne, 2008]. It introduces the idea of *virtual triples*—triples that are not matched against the underlying ontology graph, but used to configure *similarity joins*. Similarity joins specify which pair(s) of variables (that are bound to resources in *SPARQL*) should be joined and compared using a certain type of similarity measure. Thus, they establish a *virtual relation* between the resources. A similarity ontology defines the admissible virtual triples and links the different measures to their actual implementation in SimPack – our library of similarity measures.[14] For convenience we list some similarity measures used by i*SPARQL* in Table 4.1. The similarity ontology also enables the specification of more complicated combinations of similarity measures.

4.4.2 SPARQL-ML

Specifically for our bug prediction experiments in Section 4.5.6, we will use our SPARQL-ML (SPARQL Machine Learning) approach – an extension of *SPARQL* with knowledge discovery capabilities. SPARQL-ML is a tool for efficient, relational data mining on Semantic Web data.[15] Its syntax and semantics were thoroughly examined in [Kiefer et al., 2008] together with a number of case studies to show

[13] A demonstration of i*SPARQL* is available at `http://www.ifi.uzh.ch/ddis/isparql.html`.

[14] `http://www.ifi.uzh.ch/ddis/simpack.html`

[15] SPARQL-ML is available at `http://www.ifi.uzh.ch/ddis/sparql-ml.html`

Measure	Explanation
Levenshtein measure (simple)	String similarity between, for instance, class/method names: *Levenshtein* string edit distance measuring how two strings relate in terms of the number of insert, remove, and replacement operations to transform one string into the other [Levenshtein, 1966].
TreeEdit-Distance measure (simple)	Tree similarity between tree representations of classes: measuring the number of steps it takes to transform one tree into another tree by applying a set of elementary edit operations: insertion, substitution, and deletion of nodes [Sager et al., 2006].
Graph measure (simple)	Graph similarity between graph representations of classes: the measure aims at finding the maximum common subgraph (MCS) of two input graphs [Valiente, 2002]. Based on the MCS the similarity between both input graphs is calculated.
Custom-ClassMeasure (engineered)	User-defined Java class similarity measure: determines the affinity of classes by comparing their sets of method/attribute names. The names are compared by the Levenshtein string similarity measure. Individual similarity scores are weighted and accumulated to an overall similarity value.

Table 4.1: Selection of four iSPARQL similarity strategies.

the usability of SPARQL-ML. In this section we will give a brief introduction and example queries.

SPARQL-ML enables the usage of *Statistical Relational Learning (SRL)* methods such as *Relational Probability Trees (RPTs)* [Neville et al., 2003a] and *Relational Bayesian Classifiers (RBCs)* [Neville et al., 2003b] that take the relations between RDF resources into account for the induction of a model, as well as for making predictions. These methods have been shown to be very powerful for SRL as they model not only the intrinsic attributes of resources, but also the extrinsic relations to other resources [Džeroski, 2003], and thus, should perform at least as accurate as traditional, propositional learning techniques.

Listing 4.1 is the SPARQL-ML query which builds up a prediction model (indicated by the `CREATE MINING MODEL` statement). Within this block, the target variables, prediction type and feature types are defined (lines 2-5). The following block (lines 8-10) are triple patterns to bind the variables that serve as features for the prediction in the training period. Finally, in line 12, the library provid-

```
1  CREATE MINING MODEL <http://www.example.org/bugs>
2    { ?file                  RESOURCE TARGET
3      ?error                 DISCRETE PREDICT {'YES','NO'}
4      ?reportedIssues3Months CONTINUOUS
5      ?reportedIssues5Months CONTINUOUS
6    }
7  WHERE
8    { ?file rcs:hasError              ?error .
9      ?file rcs:reportedIssues3Months ?reportedIssues3Months .
10     ?file rcs:reportedIssues5Months ?reportedIssues5Months
11   }
12 USING <http://kdl.cs.umass.edu/proximity/rpt>
```

Listing 4.1: SPARQL-ML induce statement.

ing the machine learning algorithms is selected (in this case: proximity, weka would be available as well).

```
1  SELECT DISTINCT ?file ?error ?rpt ?prob
2  WHERE
3    { ?file rcs:hasError              ?error .
4      ?file rcs:reportedIssues3Months ?reportedIssues3Months .
5      ?file rcs:reportedIssues5Months ?reportedIssues5Months .
6
7      ( ?rpt ?prob ) sml:predict
8                       ( <http://www.example.org/bugs>
9                         ?file, ?error, ?reportedIssues3Months,
10                        ?reportedIssues5Months )
11   }
```

Listing 4.2: SPARQL-ML predict statement.

Listing 4.2 applies the prediction model that was learned in the query of 4.1 to a test set that is bound with the triple patterns in lines 3-5. Since the learning of a prediction model and its application to the test set are two detached queries, the learned model is passed between those two queries using a URI (line 1 in Listing 4.1 and line 8 in Listing 4.2).

4.5 Experimental Evaluation

To show the applicability and ease of use of our approach for a very broad range of Software Analysis tasks we first surveyed the last four years of the proceedings of the *ICSE Workshop on Mining Software Repositories (MSR)*[16] and then tried to replicate as many experiment types as possible with EvoOnt and the off-the-shelf query tools.

The most actively investigated software analysis tasks are shown in Table 4.2. The table shows the 12 task categories we identified together with their percentage of numbers of papers. Note that these categories were subjectively constructed. We found this procedure very useful to get an overview of current research activities, for which our Semantic Web tools could make a significant contribution. Furthermore, Table 4.2 also shows for which tasks we have successfully applied one or more of our tools.

Of the accepted 103 papers in total (not including MSR challenge reports), almost 14% are dealing with the construction and evaluation of *General Frameworks* for the integration, cleansing, analysis, and querying of data from various software-development related sources, such as versioning and bug-tracking systems, source code, forums, mailing lists, etc. Our EvoOnt approach *is*, in fact, a unified, general purpose framework integrating software data from diverse sources and enabling its efficient querying and analysis along a multitude of dimensions.

Approximately the same number of papers investigate the task of *Bug and Change Prediction* to find the locations in software that most likely will have to be fixed in the future based on historical

[16] `http://www.msrconf.org/`

[17] These tasks could theoretically be accmplished with SPARQL. However, we did not conduct any social network analysis experiments due to the lack of datasets.

Task	Fraction (%)	Domain Independent Tool
General Framework (*e.g.*, facilitate analysis process, data cleansing & integration, repository query language)	13.59	RDF, OWL, *SPARQL*
Bug/Change Prediction (*e.g.*, build defect detectors/classifiers, bug risk & fixing time prediction)	13.59	SPARQL-ML
Social Network Analysis (*e.g.*, mailing list analysis, understand developer roles & networks, discover development processes)	11.65	—[17]
Software Evolution Analysis (*e.g.*, study & characterize system evolution, visualization)	10.68	i*SPARQL*
Software Reuse (*e.g.*, code suggestion, similarity analysis, code search & retrieval, clone detection)	10.68	*SPARQL*, i*SPARQL*
Mining CVS (*e.g.*, mine local histories)	9.71	*SPARQL*
Change Impact Analysis (*e.g.*, detect incomplete refactorings, signature change analysis, code smells)	9.71	*SPARQL*
General Mining (*e.g.*, find sequences of changed files)	8.74	*SPARQL*
Text Mining (*e.g.*, free text search, mining code comments, keyword search)	4.85	*SPARQL*, i*SPARQL*
Source Code Metrics (*e.g.*, code clone coverage)	2.91	*SPARQL*
Repository Mining Tools (*e.g.*, evaluation of tools)	1.94	—
Pattern Detection (*e.g.*, detect software design patterns, find system-user interaction patterns)	1.94	*SPARQL*
	100% (103 papers)	

Table 4.2: Popular software analysis tasks from MSR 2004 – 2007

information. This is a perfect candidate for our SPARQL-ML tool as it allows us to make a statistical statement about the likelihood of the occurrence of bugs and changes in source code (see Section 4.5.6).

Another set of 12 papers examines methods from *Social Network Analysis* to, for instance, determine developer roles and to reveal software development processes. We did not yet address any of these tasks with one of our tools. This is not a limitation of our approach and the used techniques themselves but of the data sets available to us. We believe that our tools could be applied to these tasks with comparable performance.

Software Evolution Analysis and *Software Reuse* are the fourth and fifth largest categories. These categories are interesting as they hold tasks such as *evolution visualization*, *similarity analysis*, as well as *code search & retrieval* that can clearly be tackled by our Semantic Web approaches.

Additional categories we found suitable for further consideration are *Change Impact Analysis*, *Source Code Metrics*, and *Pattern Detection*. Specifically, the first one includes *detection of code smells* (*i.e.*, code design flaws) that can partly be solved by approaches falling into the second category to compute *source code metrics*. *Pattern Detection* is in range of our tools as our FAMIX-based software model approach allows us to query the RDF data set for certain *software design patterns*.

Note that tasks such as *visualization* and *search* are common to almost all categories. We address visualization in Section 4.5.2, in which we apply i*SPARQL* to discover and visualize the architectural evolution of software components.

Given these categories, we chose to conduct the following five sets of experiments (in increasing order of complexity):

1. *software evolution measurements*: analyzing and visualizing changes between different releases;
2. *metrics experiments*: evaluation of the ability to calculate object-oriented software metrics;
3. *impact experiments*: evaluation of the applicability of Semantic Web tools to detect code smells;
4. *density measurements* (as a subtask of the evolution and metrics experiments): determining the amount of bug-fixing and "ordinary" software development measured over all software engineering activities;
5. *bug prediction assessments*: showing the usefulness of SPARQL-ML for bug prediction.

4.5.1 Experimental Setup and Datasets

For our experiments, we examined 206 releases of the `org.eclipse.compare` plug-in for Eclipse. This plug-in consists in average of about 150 java classes per version. Multiplied with the 206 releases we have the source code information of roughly 30'000 classes in our repository[18]. To generate an OWL data file of a particular release, it was first automatically retrieved from Eclipse's CVS repository and loaded into an in-memory version of our software ontology model, before it was exported to an OWL file. To get the data from CVS and to fill our version ontology model, the contents of the Release History Database (RHDB) [Fischer et al., 2003] for the compare plug-in were loaded into memory and, again, parsed and exported to OWL

[18] We believe that lines of code is not a suitable metric in this cases because we use graph-based representations. However, for comparison, the LOC of one version is about 38'000. Multiplied with the 206 versions we have information about approximately 7.8 millions LOC.

according to our version ontology model. While parsing the CVS data, the commit message of each revision of a file was inspected and matched against a regular expression to detect referenced bug IDs. If a bug was mentioned in the commit message as, for instance, in *"fixed #67888: [accessibility] Go To Next Difference stops working on reuse of editor"*, the information about the bug was (automatically) retrieved from the web and also stored in memory. Finally, the data of the in-memory bug ontology model was exported to OWL. None of the above steps needed any kind of user interaction (except for selecting the project and versions to export) and were conducted by an Eclipse plug-in allowing us to rely on a multitude of functions provided by the Eclipse framework such as checkout of a release or build-up and traversal of the syntax trees. A general downside of the design of our extraction tool was the generation of an in-memory model before we wrote the data to RDF/OWL. This fact limited us in the choice of project sizes because the in-memory models of projects larger than 150 classes per version reached the limit of the physical main memory of the extracting machine. A currently developed 2nd version of the extraction tools now directly generates triples that get immediately written to disk. Therefore, the project size ceases to be a limiting factor.

Recently, Gröner *et al.* [Gröner et al., 2008] compared query approaches in reverse engineering. Specifically, they compared GUPRO/GReQL with OWL/SPARQL and showed that the time costs are, in summary, more than ten times higher for OWL/SPARQL than for GUPRO/GReQL. Note that this performance difference needs to be seen in the light of our other investigations, where we showed that simple selectivity-based query optimization applied to existing SPARQL engines techniques can lead to performance improvements of 3-4 orders of magnitude [Bernstein

et al., 2007b, Stocker et al., 2008]. In particular, since Gröner *et al.* found that KAON2 was about 1 order of magintude slower than GUPRO/GReQL and we found that our static query optimizer sped up typical SPARQL queries on KAON2 by about 600 times (compared to other SPARQL engines even by about 700 times) [Bernstein et al., 2007b], we can expect that optimized SPARQL engines should provide an at least equal if not superior performance compared to GUPRO/GReQL. As a consequence, we share Gröner *et al.* 's opinion that recent research [Weiss et al., 2008, Abadi et al., 2007] will lead to vast improvements in execution-time of SPARQL queries, and, therefore, our approach will most probably have a competitive time complexity in future applications.

4.5.2 Task 1: Software Evolution Analysis

With the first set of experiments, we wanted to evaluate the applicability of our i*SPARQL* approach to the task of software evolution visualization (*i.e.*, the graphical visualization of code changes for a certain time span in the life cycle of the Eclipse `compare` software project). This analysis is especially important when trying to detect code clones. To that end, we compared all the Java classes of one major release with all the classes from another major release with different similarity strategies mirroring the experiments of Sager et al. [Sager et al., 2006] Listing 4.3 shows the corresponding query for two particular releases and the *Tree Edit Distance* measure. In lines 3-6 and 8-11 of Listing 4.3, four variables are bound to each class URI and its literal value. Identified by the `IMPRECISE` keyword, each class URI is passed to a property function (`isparql:treeEditDistance`, line 14) which binds the calculated structural similarity to the variable `sim1`. Another similarity algorithm is applied to the class names (`isparql:levenshtein`,

line 15). Finally, the two similarities are weighted and combined to an overall score (line 16).

```
 1  SELECT ?similarity
 2  WHERE
 3    { ?class1   som:uniqueName ?name1 ;
 4                som:isClassOf  ?file1 .
 5      ?file1    som:hasRelease ?release1 .
 6      ?release1 vom:name       ''R3_1'' .
 7
 8      ?class2   som:uniqueName ?name2 ;
 9                som:isClassOf  ?file2 .
10      ?file2    som:hasRelease ?release2 .
11      ?release2 vom:name       ''R3_2'' .
12
13      IMPRECISE
14        { ?sim1       isparql:treeEditDistance ( ?class1 ?class2 ).
15          ?sim2       isparql:levenshtein      ( ?name1 ?name2 ).
16          ?similarity isparql:score
    ( 0.25 ?sim1 0.75 ?sim2 )
17        }
18    }
19  ORDER BY DESC (?similarity)
```

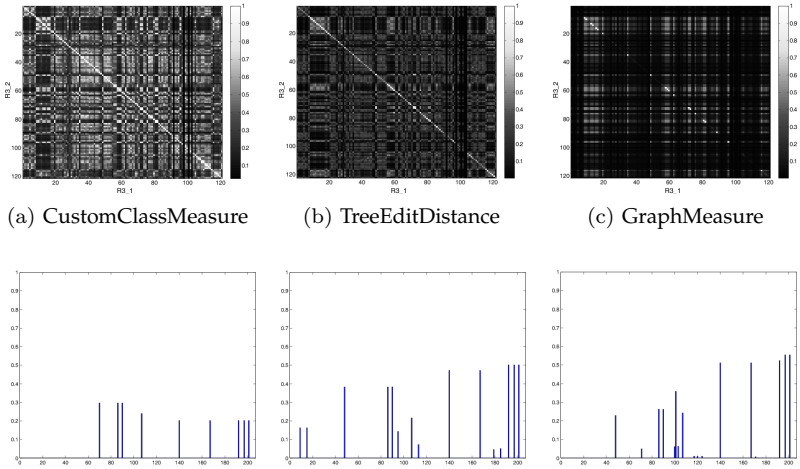
Listing 4.3: iSPARQL query: Computation of the structural (Tree Edit Distance) and textual (Levenshtein) similarity between the classes of two releases.

The results of the execution of Listing 4.3 for the releases 3.1 and 3.2 are shown in Figure 4.2. The heatmaps mirror the class code changes between the two releases of the project by using different shades of gray for different similarity scores in the interval $[0, 1]$. Analyzing the generated heatmaps, we found that the specialized *Custom Class Measure* performed best for the given task; most likely, this is because it is an algorithm especially tailored to compare source code classes. The combination of method/attribute set comparisons together with the *Levenshtein* string similarity measure for method-/attribute names (Figure 4.2b) turned out to be less precise. In all our experiments, the *Graph Measure* (Figure 4.2c) was the least accurate indicator for the similarity of classes. What is common to

Figures 2(a–c) is the diagonal line denoting high similarity of the same classess between different versions. This is an obvious and expected fact because usually only a small percentage of the source code changes between two versions. Another, less obvious fact is the high similarity observed in the top-left area of the figures. This is a cluster of classes very similar to each other, but highly different to the rest of the classes. An in-depth analysis showed that this cluster consists of interface definitions, which lack a lot of features of "normal" classes (e.g. method bodies, variable declarations, anonymous classes). In general, a software project manager or auditor can use the information of these visualizations to get a preselection of possible candidates for duplicate code.

Furthermore, to shed some light on the history of a single Java class, we measured the similarity of the class from one release and the (immediate) next release and repeated this process for all classes and releases. This resulted in an array of values $sim_{class}^{R_i,R_j}$, each value expressing the similarity of the same class of two different releases R_i and R_j. However, to visualize the *amount of change*, we plotted the inverse (*i.e.*, $1 - sim_{class}^{R_i,R_j}$) as illustrated in Figures 2(d–f) that show the history of changes for three distinct classes of the project. There are classes such as `BufferedCanvas` which tend to have fewer changes as the project evolves over time.

Other classes such as `CompareEditor` (Figure 4.2e) are altered again and again, probably implying some design flaws or code smells. Then again, there are classes which tend to have more changes over time as shown in Figure 4.2f for the class `Utilities`. This information can also help to manage a software project since it allows managers to allocate resources to classes/components that have a high development activity.



(a) CustomClassMeasure (b) TreeEditDistance (c) GraphMeasure

(d) BufferedCanvas (e) CompareEditor (f) Utilities

Figure 4.2: Figures (a)–(c) depict the computed heatmaps of the between-version comparison of all the classes of releases 3.1 and 3.2 of the `org.eclipse.compare` plugin using three different similarity strategies. Furthermore, the history of changes for three distinct classes of the project is illustrated in Figures (d)–(f).

4.5.3 Task 2: Computing Source Code Metrics

With our second set of experiments, we wanted to demonstrate the possibility of calculating *object-oriented software design metrics* [Lanza and Marinescu, 2006] using *SPARQL*. For illustration purposes, we have chosen six of them which we will succinctly discuss in this section. Note that there is a close connection between code smells and software metrics in the sense that metrics are often used to identify possible code smells in object-oriented software systems (see Section 4.5.4).

Changing Methods (CM) and Changing Classes (CC)

When changing the functionality of a method (*callee*), in most cases this will have an impact to the invoker (*caller*) of this method. Consider a method in an arbitrary class for sorting a list of input strings. For some reason a developer might decide to change the order of the result list, for instance, from ascending to descending order. If the change is not communicated to the caller, it might be unable to process the results correctly as it expects the returned list to be sorted in ascending order.

This is a typical example for a change that can lead to defects of the invoking methods and classes as they expect different semantics of the callee. Therefore, a method that is invoked by many other methods has a higher risk of causing a defect because a developer might forget to change every invoking method.

With *SPARQL* we are able to easily count the number of times a method is called. This is shown in Query 4.4 which lists methods and the number of their distinct callers (variable ?cm; in query). In addition, the query counts the number of distinct classes these calling methods are defined in (variable ?cc; also in query). The topmost answers of the query are shown in Table 4.3.

```
1  SELECT ?method (count(distinct ?invoker) AS ?cm)
2                 (count(distinct ?invokerClass) AS ?cc)
3  WHERE
4    { ?class        som:hasMethod ?method .
5      ?invoker      som:invokes   ?method .
6      ?invokerClass som:hasMethod ?invoker
7    }
8  GROUP BY ?method
9  ORDER BY ASC(?method)
```

Listing 4.4: Changing methods (CM)/classes (CC) query pattern.

class	method	cm	cc
CompareUIPlugin	getDefault()	30	10
Utilities	getString(java.util.ResourceBundle,java.lang.String)	26	14
Utilities	getString(java.lang.String)	24	12
ICompareInput	getLeft()	16	9
ICompareInput	getRight()	15	8

Table 4.3: Changing methods/classes for the `compare` plug-in

Number of Methods and Number of Attributes

The queries shown in Listing 4.5 and 4.6 calculate the two metrics *number of attributes (NOA)* and *number of methods (NOM)* that can be both used as indicators for possible *God classes* (see also Section 4.5.4). The results are shown in Table 4.5a and 4.5b respectively. Having a closer look at class `TextMergeViewer`, one can observe that the class is indeed very large with its 4344 lines of code. Also `CompareUIPlugin` is rather big with a total number of 1161 lines of code. Without examining the classes in more detail, we hypothesize that there might be some room for refactorings, which possibly result in smaller and more easy to use classes.

```
1 SELECT ?class (count(distinct ?attribute) AS ?noa)
2 WHERE
3   { ?class som:hasAttribute ?attribute }
4 GROUP BY ?class
5 ORDER BY ASC(?class)
```

Listing 4.5: Number of attributes (NOA) query pattern.

```
1 SELECT ?class (count(distinct ?method) AS ?nom)
2 WHERE
3   { ?class som:hasMethod ?method }
4 GROUP BY ?class
5 ORDER BY ASC(?class)
```

Listing 4.6: Number of methods (NOM) query pattern.

class	noa
TextMergeViewer	91
PatchMessages	63
CompareUIPlugin	42
ContentMergeViewer	36
CompareMessages	27
EditionSelectionDialog	26
CompareEditorInput	23
CompareConfiguration	20
ICompareContextIds	19
ComparePreferencePage	18

(a) Number of attributes (NOA) metric for the `compare` plug-in.

class	nom
TextMergeViewer	115
CompareUIPlugin	46
ContentMergeViewer	44
OverlayPreferenceStore	43
Patcher	39
CompareEditorInput	38
Utilities	34
MergeSourceViewer	31
EditionSelectionDialog	30
CompareConfiguration	28

(b) Number of methods (NOM) metric for the `compare` plug-in.

Table 4.4: The results of NOA and NOM queries.

NUMBER OF BUGS (NOB) AND NUMBER OF REVISIONS (NOR) To close this section and to support or discard our hypothesis from the previous paragraph, we measured the *number of bug reports (NOB)* issued per class as we assume a relationship between the number of class methods (attributes) and the number of filed bug reports. To that end, we executed a query (not shown) similar to the one presented in Listing 4.6. Indeed, there is a relationship as the results in Table 4.6a clearly show: the two classes `TextMergeViewer` and `CompareUIPlugin` are also among the top 10 of the most buggiest classes in the project.

Finally, the *number of revisions (NOR)* metric counts the number of revisions of a file recorded in CVS. The respective results are shown in Table 4.6b (again, for space consideration we omitted the listing of the query. However, it is very similar to Listing 4.6). Both NOB and NOR are used in Section 4.5.5 to determine *defect* and *evolution density* of software systems.

file	nob
TextMergeViewer	36
CompareEditor	16
Patcher	15
PreviewPatchPage	13
ResourceCompareInput	12
DiffTreeViewer	10
Utilities	10
CompareUIPlugin	9
StructureDiffViewer	9
PatchWizard	6

(a) Number of bug reports for the `compare` `compare` plug-in.

file	nor
TextMergeViewer	213
CompareEditorInput	88
CompareUIPlugin	70
ContentMergeViewer	69
EditionSelectionDialog	66
Utilities	64
CompareEditor	57
Patcher	51
ComparePreferencePage	50
DiffTreeViewer	47

(b) Number of revision for the `compare` plug-in.

Table 4.5: The results of NOB and NOR queries.

4.5.4 Task 3: Detection of Code Smells

In a third set of experiments, we evaluate the applicability of *SPARQL* to the task of detecting *code smells* [Fowler, 1999]. In other words, the question is whether *SPARQL* is able to give you a *hint* that there *might* be potential problems in the source code. Can *SPARQL* tell you if it could be solved, for instance, by refactoring the current architecture? In order to solve this task, we selected one candidate smell, the *GodClass* anti-pattern, which we thought could be (among others) identified in the `compare` plug-in.

Furthermore, the following experiments are useful to demonstrate the benefits of *ontological reasoning* for software analysis. We, therefore, use two well-known inference engines in the Semantic Web: the Jena[19] reasoner and the complete OWL reasoner Pellet.[20] Note that, while the Jena reasoner only supports a subset of the OWL language (*i.e.*, OWL/Lite), Pellet is complete, in other words is able to deal with all elements of the OWL/DL language. These reasoners are used to derive additional RDF assertions which are entailed from

[19] `http://jena.sourceforge.net/`
[20] `http://pellet.owldl.com/`

base facts together with the ontology information from the source code models (Section 4.3) and the axioms and rules associated with the reasoners.

GodClass Anti-Pattern

A God class is defined as a class that potentially ''knows" too much (its role in the program becomes all-encompassing). In our sense, it has (too) many methods and instance variables. In the following, we present two approaches to find God classes in source code: first, by computing object-oriented source code metrics (see Section 4.5.3) and, second, by inferring them using the aforementioned reasoning engines. To illustrate our approach, we define a God class as any class which declares more than 20 methods and attributes in its body.

We first present the metrics-based approach. Listing 4.7 shows a particular *SPARQL* query that counts both the number of methods (NOM) and number of attributes (NOA) per class. A God class is successfully identified if both are above 20. The topmost results of this query are shown in Table 4.6.

```
1  SELECT ?GodClass (count(distinct ?method) AS ?nom)
2                   (count(distinct ?attribute) AS ?noa)
3  WHERE
4    { ?GodClass som:hasMethod    ?method ;
5                som:hasAttribute ?attribute
6    }
7  GROUP BY ?GodClass
8  HAVING ( ( count(distinct ?method) > 20 )
9         && ( count(distinct ?attribute) > 20 ) )
10 ORDER BY ASC(?GodClass)
```

Listing 4.7: SPARQL metrics approach to find God classes.

To demonstrate how God classes can be inferred from the ontology, the *ontological concept 'GodClass'* has to be defined first. We chose

GodClass	nom	noa
TextMergeViewer	115	91
CompareUIPlugin	46	42
ContentMergeViewer	44	36
CompareEditorInput	38	23
EditionSelectionDialog	30	26

Table 4.6: Results of God class query pattern.

OWL as concept definition language as it offers all the required language constructs.

Figure 4.3 shows the definition of the ontology concept 'GodClass' in description logic syntax (DL Syntax) wich can be easily transformed to OWL syntax, *e.g.*, N3. We define a new class (`GodClass`) which is equivalent to an anonymous class of the type `som:Class` having at least 21 `hasMethod` and 21 `hasAttribute` relations.

$$\text{smell:GodClass} \equiv \text{som:Class} \sqcap \geqslant 21\ \text{som:hasMethod} \sqcap \geqslant 21\ \text{som:hasAttribute}$$

Figure 4.3: GodClass concept definition (in DL Syntax).

Having defined the ontological concept for a GodClass and, of course, an inferred ontology created by a reasoner, it is now possible to use the query shown in Listing 4.8 to find all God classes in the `compare` plug-in.

```
1 SELECT ?GodClass
2 WHERE
3   { ?GodClass a smell:GodClass }
```

Listing 4.8: SPARQL reasoning approach to find God classes.

Orphan Methods

To close this section, we give an example where ontological reasoning is not successful although the concept can be perfectly defined in OWL. Figure 4.4 shows the logical definition for *orphan methods* (*i.e.*, methods that are not invoked by any other method in the project). The ¬ expression describes a logical negation: the class extension consists of those methods that are not invoked by any other behavioral entity (*i.e.*, any other method).

Due to the *open-world semantics* of OWL that states that if a statement cannot be inferred from the RDF data set, then it still cannot be inferred to be false, most inference engines, including the ones used in this work, are not able to find concepts of type `OrphanMethod`.

Therefore, the query shown in Listing 4.9 does not return any results. Fortunately, there is a trick one can do in *SPARQL* queries to get a little bit of *closed-world reasoning* — the ability to answer *negative* queries although the RDF data set does not contain explicit information about the absence of certain facts.

$$\text{smell:OrphanMethod} \equiv \text{som:Method} \sqcap \neg\text{som:isInvokedBy.BehaviouralEntity}$$

Figure 4.4: Orphan method concept definition (in DL Syntax).

```
1  SELECT ?orphanMethod WHERE
2    { ?orphanMethod a smell:OrphanMethod }
```

Listing 4.9: Orphan method query pattern.

The trick is achieved by the `bound` operator in the filter clause on line 6 in Listing 4.10, which returns true if its variable (`?invoker`)

is bound to a value. The query in Listing 4.10 finds all `?orphanMethod`s, gets any `isInvokedBy`, and filters those which passed through the optional branch.

```
1  SELECT ?orphanMethod WHERE
2    { ?orphanMethod rdf:type som:Method .
3      OPTIONAL
4        { ?orphanMethod som:isInvokedBy ?invoker }
5      FILTER ( ! bound(?invoker) )
6    }
```

Listing 4.10: Orphan method query pattern.

However, some ontological reasoning is still required in this query, as the property `isInvokedBy` is defined as `owl:inverseOf invokes` in our software ontology model. In other words, results of the form *method1 isInvokedBy method2* must be inferred from the inverse `invokes`-statements.

The query returns numerous results of which we only present one. It finds, for instance, the public method `discardBuffer()` declared on class `BufferedContent`. This method is never invoked by any other class in the `compare` plug-in. Orphan methods could possibly be removed from the interface of a class without affecting the overall functionality of the system to result in a more clean and easy to understand source code.

4.5.5 Task 4: Defect and Evolution Density

With our next set of experiments, we aim at determining a file's as well as a whole software project's *Defect* and *Evolution Density*. Note that in this context, we consider files as "containers" for classes and instance variables (*i.e.*, they may contain multiple classes as well as inner classes). Inspired by Fenton [Fenton, 1991], defect density

DED_f of a file f is defined as the ratio of the number of bug reports (NOB) over the total number of revisions (NOR) of f, *i.e.*,

$$DED_f = \frac{NOB}{NOR} \tag{4.1}$$

where NOB and NOR are the metrics presented in Section 4.5.3. Next, we define a file's/project's *Evolution Density* as counterpart to defect density. When we refer to evolution density, we think of all the changes made to a software system which were not bug-fixing, but "ordinary" software development, such as functional extension and improvement, adaption, and testing. The evolution density EVD_f of a file f is, therefore, defined as:

$$EVD_f = 1 - DED_f \tag{4.2}$$

Table 4.7 lists evolution and defect density for the 5 topmost classes of the `org.eclipse.compare` plug-in in descending order of defect density retrieved with the query shown in Listing 4.11. Visualizing the defect density (Figure 4.5a) brings to light some interesting facts: first, only about 25% of all source files contain bugs at all. Nearly 75% of the code is free of defects (measured by the reported bugs); second, the concentration of the errors is exponentially decreasing (*i.e.*, only few files have a high concentration of bugs). This is further illustrated in Figure 4.5b, which shows a histogram of the number of classes in the project per 0.1 DED interval.

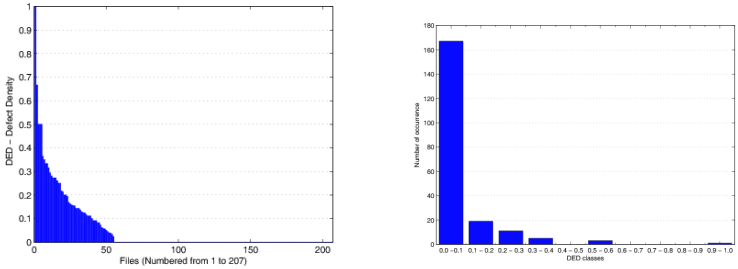
Finally, to calculate measures *over all software engineering activities* in the project, *Total Evolution Density* ($TEVD$) and *Total Defect Den-*

```
SELECT ?fileName (count(?revision) AS ?NOR)
  (count(?bug) AS ?NOB)
  (count(?bug)/count(?revision) AS ?DED)
  (1-count(?bug)/count(?revision) AS ?EVD)
WHERE {
  ?file vom:hasRevision ?revision .
  ?file vom:name ?fileName .
  OPTIONAL{
    ?bug bom:hasResolution ?revision .
  }
  FILTER(regex(?fileName, "\\.java$" , "i")) .
  }
  GROUP BY (?fileName)
```

Listing 4.11: Evolution and density query pattern.

File	NOR	NOB	EVD	DED
StatusLineContributionItem.java	3	3	0.000	1.000
CompareNavigator.java	3	2	0.333	0.667
IResourceProvider.java	4	2	0.500	0.500
DifferencesIterator.java	10	5	0.500	0.500
PatchProjectDiffNode.java	2	1	0.500	0.500

Table 4.7: Evolution and defect density of the `org.eclipse.compare` plug-in.



(a) DED per file. (b) DED histogram.

Figure 4.5: The figures show the defect density *DED* per file and the number of classes per 0.1 DED interval in the `org.eclipse.compare` plug-in release 3.2.1.

sity ($TDED$) are defined as shown in Equations 4.3 and 4.4 (with n being the number of files).

$$TEVD = \frac{\sum_{f=1}^{n} EVD_f}{n} \tag{4.3}$$

$$TDED = \frac{\sum_{f=1}^{n} DED_f}{n} = 1 - TEVD \tag{4.4}$$

For the `org.eclipse.compare` plug-in release 3.2.1, the value for $TDED$ is 0.054, which expresses that 5.4% of all activities in the project are due to bug-fixing and 94.6% due to functional extension (among others). These findings seem to disagree with those of Boehm [Boehm, 1981] who found that approximately 12% of all software engineering tasks are bug-fixing. We hypothesize that the *time span* of the measurements and the *bug reporting discipline* are reasons for this divergence in results and postpone it to future work to prove or reject this hypothesis.

4.5.6 Task 5: Bug Prediction

For our final bug prediction experiments, we will use our SPARQL-ML approach (*SPARQL* Machine Learning) – an extension of *SPARQL* that extends the Semantic Web query language with knowledge discovery capabilities (see Section 4.4.2). In order to show the usefulness of SPARQL-ML for bug prediction, we repeated the *defect location experiment* presented in [Bernstein et al., 2007a]. The goal of this experiment was to predict the probability of defect (bug) occurrence for any given file from a test set given an induced model from a training set. The data for the experiment was collected from six plug-ins of the Eclipse open source project as in [Bernstein et al.,

2007a]: `updateui`, `updatecore`, `search`, `pdeui`, `pdebuild`, and `compare`.

The experimental procedure can be summarized as follows: first, along with the data from CVS and Bugzilla, we exported each plug-in into our Semantic Web EvoOnt format [Kiefer et al., 2007c][21]; second, providing a small extension to EvoOnt, we took into account the extra features from [Bernstein et al., 2007a] that are used for learning and predicting; and third, we wrote SPARQL-ML queries for the induction of a mining model on the training set as well as for the prediction of bugs on the test set. The queries for both tasks are shown in Listings 4.12 and 4.13 respectively.

In the past, many approaches have been proposed to perform bug prediction in source code. In Fenton and Neil [Fenton and Neil, 1999], an extensive survey and critical review of the most promising learning algorithms for bug prediction from the literature is presented. They proposed to use *Bayesian Belief Networks (BBNs)* to overcome some of the many limitations of the reviewed bug prediction algorithms. It is imporatant to note that the relational Bayesian classifier (RBC) validated in this case study is an extension of the naïve Bayesian classifier (that applies Bayes' rule for classification) to a relational data setting.

The results are illustrated in Figure 4.6, showing the results in terms of prediction accuracy (acc; in legend), Receiver Operating Characteristics (ROC; graphed), and the area under the ROC-curve (auc; also in legend). The ROC-curve graphs the true positive rate (y-axis) against the false positive rate (x-axis), where an ideal curve would go from the origin (0,0) to the top left (0,1) corner, before proceeding to the top right (1,1) one [Provost and Fawcett, 2001]. It has

[21] Information from CVS and Bugzilla was considered from the first releases up to the last one released in January 2007.

```
1  CREATE MINING MODEL <http://www.example.org/bugssignificant>
2    { ?file                    RESOURCE    TARGET
3      ?error                   DISCRETE    PREDICT  {'YES','NO'}
4      ?lineAddedIRLAdd         CONTINUOUS
5      ?lineDeletedIRLDel       CONTINUOUS
6      ?revision1Month          CONTINUOUS
7      ?defectAppearance1Month  CONTINUOUS
8      ?revision2Months         CONTINUOUS
9      ?reportedIssues3Months   CONTINUOUS
10     ?reportedIssues5Months   CONTINUOUS
11   }
12 WHERE
13   { ?file      vom:hasRevision   ?revision .
14     ?revision  vom:creationTime  ?creation .
15     FILTER (
16     xsd:dateTime(?creation) < "2007-01-31T00:00:00"^^xsd:dateTime)
17
18     ?file      vom:hasError  ?error .
19
20     OPTIONAL { ?file  vom:lineAddedIRLAdd
   ?lineAddedIRLAdd . }
21     OPTIONAL { ?file  vom:lineDeletedIRLDel
   ?lineDeletedIRLDel . }
22     OPTIONAL { ?file  vom:revision1Month
   ?revision1Month . }
23     OPTIONAL { ?file  vom:defectAppearance1Month
   ?defectAppearance1Month . }
24     OPTIONAL { ?file  vom:revision2Months
   ?revision2Months . }
25     OPTIONAL { ?file  vom:reportedIssues3Months
   ?reportedIssues3Months . }
26     OPTIONAL { ?file  vom:reportedIssues5Months
   ?reportedIssues5Months . }
27   }
28 USING <http://kdl.cs.umass.edu/proximity/rpt>
```

Listing 4.12: SPARQL-ML model induce statement.

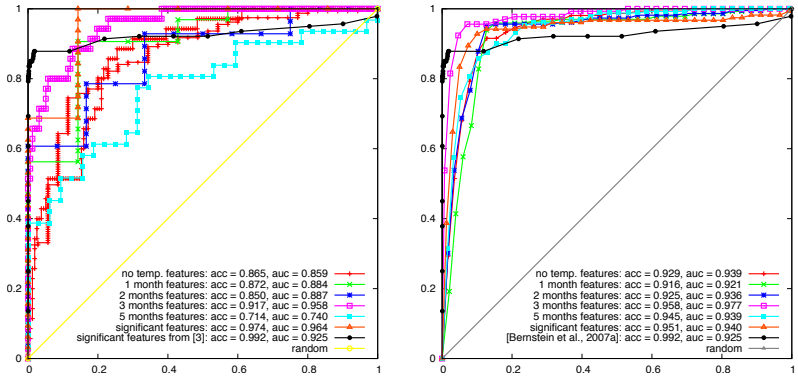
the advantage to show the prediction quality of a classifier independent of the distribution of the underlying data set (e.g. the skewed ratio between bug and no-bug). The area under the ROC-curve is, typically, used as a summary number for the curve. An in-depth explanation about ROC-curves can be found in [Witten and Frank, 2005]. Note that this experiment clearly illustrates the simplicity by which the experiment from [Bernstein et al., 2007a] can be reduced to running an off-the-shelf query.

```
1  SELECT DISTINCT ?file ?prediction ?probability
2  WHERE
3    { ?file      vom:hasRevision   ?revision .
4      ?revision  vom:creationTime  ?creation .
5
6      FILTER (
7      xsd:dateTime(?creation) <= "2007-01-31T00:00:00"^^xsd:dateTime)
8
9      OPTIONAL { ?file  vom:lineAddedIRLAdd
   ?lineAddedIRLAdd . }
10     OPTIONAL { ?file  vom:lineDeletedIRLDel
   ?lineDeletedIRLDel . }
11     OPTIONAL { ?file  vom:revision1Month
   ?revision1Month . }
12     OPTIONAL { ?file  vom:defectAppearance1Month
   ?defectAppearance1Month . }
13     OPTIONAL { ?file  vom:revision2Months
   ?revision2Months . }
14     OPTIONAL { ?file  vom:reportedIssues3Months
   ?reportedIssues3Months . }
15     OPTIONAL { ?file  vom:reportedIssues5Months
   ?reportedIssues5Months . }
16
17     PREDICT
18       { ( ?prediction ?probability )
19        sml:predict  (
20        <http://www.example.org/bugssignificant>
21        ?file ?lineAddedIRLAdd ?lineDeletedIRLDel
22        ?revision1Month ?defectAppearance1Month ?revision2Months
23        ?reportedIssues3Months ?reportedIssues5Months ) .
24       }
25   }
```

Listing 4.13: SPARQL-ML predict statement.

4.6 Conclusions, Limitations, and Future Work

In this paper, we presented a novel approach to analyze software systems using Semantic Web technologies. As exemplified by the case studies above EvoOnt provides the basis for representing software source code and meta-data in OWL. This representation allows to reduce many mining software repository tasks to simple queries in the Semantic Web query language SPARQL (and its extensions i*SPARQL* and SPARQL-ML).



(a) RPT performance. (b) RBC performance.

Figure 4.6: ROC-curves to show a performance comparison of the two classifiers Relational Probability Tree (RPT) and Relational Bayesian Classifier (RBC).

This format is principally used within the Semantic Web to share, integrate, and reason about data of various origin. We evaluated the use of the format in the context of analyzing the compare plug-in for Eclipse (`org.eclipse.compare`).

To illustrate the power of using EvoOnt we conducted five sets of experiments in which we showed, firstly, that it was expressive enough to shed some light on the evolution of software systems (especially when using i*SPARQL* and its imprecise querying facilities); secondly, that it allowed to find code smells, hence, fosters refactoring; thirdly, that it enables the easy application of software design metrics to quantify the size and complexity of software; forth, that it, due to OWL's ontological reasoning support, furthermore allows to derive additional assertions, which are useful for software engineering tasks; and fifth, that it enables defect and evolution density mea-

surements expressing the amount of bug-fixing and "ordinary" software development as measured by all software engineering tasks.

A limitation of our approach is the loss of information due to the use of our FAMIX-based software ontology model. Language constructs such as switch-statements are not modeled in our ontology. Hence, the effects are that measurements on the statement level of source code cannot be conducted. This limitation can be addressed by adding elements from additional meta-models such as DMM [Lethbridge et al., 2004].

Also, one of the greatest impediments towards the widespread use of EvoOnt is the current lack of high-performance industrial-strength triple-stores and reasoning engines. Without such engines most software developers are likely to retort to relational storage solutions that are ill-suited for storing these graph-like data [Abadi et al., 2007, Weiss et al., 2008]. Some newer developments both in industry[22] and in academe [Abadi et al., 2007, Weiss et al., 2008] are encouraging. They indicate that fast engines are possible and likely to become more available in the near future. With their advent the widespread use of the techniques proposed here will become feasible and attractive to software developers.

In summary, we have shown that the use of EvoOnt can simplify a large number of software engineering tasks attempted by the mining software repositories community. We think that approaches like EvoOnt have an even greater potential as they would foster more exchange leading to a better integration of results between different analyses or simplifying inter-software-project software analyses — a problem so far avoided by most researchers due to the complexities of integrating the data. Also, the choice of the OWL as the underlying knowledge representation simplifies the extension of the model

[22] *e.g.* AllegroGraph, `http://agraph.franz.com/allegrograph/`

with other sources such as data extracted mailing lists or run-time properties of the code.

Acknowledgements

We would like to thank the participants of the 2007 ICSE Mining Software Repositories workshop and the 2007 ESWC Semantic Web Software Engineering Workshops for their valuable comments on our earlier work. We would also like to thank the anonymous reviewers for their invaluable comments that helped improve the paper.

5

Applied Temporal RDF: Efficient Temporal Querying of RDF Data with SPARQL

Jonas Tappolet and Abraham Bernstein

Published in the *Proceedings of the 6th European Semantic Web Conference, 2009*

ABSTRACT Many applications operate on time-sensitive data. Some of these data are only valid for certain intervals (e.g., job-assignments, versions of software code), others describe temporal events that happened at certain points in time (e.g., a person's birthday). Until recently, the only way to incorporate time into Semantic Web models

was as a data type property. Temporal RDF, however, considers time as an additional dimension in data preserving the semantics of time.

In this paper we present a syntax and storage format based on named graphs to express temporal RDF. Given the restriction to pre-existing RDF-syntax, our approach can perform any temporal query using standard SPARQL syntax only. For convenience, we introduce a shorthand format called τ-SPARQL for temporal queries and show how τ-SPARQL queries can be translated to standard SPARQL. Additionally, we show that, depending on the underlying data's nature, the temporal RDF approach vastly reduces the number of triples by eliminating redundancies resulting in an increased performance for processing and querying. Last but not least, we introduce a new indexing approach method that can significantly reduce the time needed to execute time point queries (e.g., what happened on January 1st).

5.1 Introduction

Time, intervals, and versioning are central aspects of many applications. People in organizations, for example, hold many different positions over time, software, comes in many revisions, or services have temporal constraints (such as guaranteed executions times). As a consequence, there exist many data representations of time. Today's approaches, however, code temporal information as additional data inside the data model. Therefore, temporal information is implicit in the data and only difficult to access by programs. Indeed, the semantics of the temporal information needs to be completely coded in the client programs accessing the data. The Semantic Web, however, claims the accessibility of semantics by machines

/ agents and, therefore, an explicit time representation should support and preserve this central pillar of the Semantic Web.

In this paper we propose to use time as an additional semantic dimension of data. Therefore, it needs to be regarded as an element of the meta model instead of being just part of the data model. By applying the foundations established by Gutierrez et al.[Gutierrez et al., 2007, Gutierrez et al., 2005], our approach proposes to implement, first, an RDF [Klyne and Carroll, 2004b] compatible syntax for temporal data. Second, we provide strategies for efficiently storing temporal RDF data (including an interval based index structure). Last but not least, we introduce a temporal extension to SPARQL [Prud'hommeaux and Seaborne, 2008] called τ-SPARQL and show how τ-SPARQL queries can be automatically translated to standard SPARQL operating on our storage scheme leveraging existing RDF/SPARQL infrastructure.

The paper first discusses the relevant related work. Then, it covers the proposed syntax for temporal RDF and τ-SPARQL. After presenting the mapping of τ-SPARQL to SPARQL it introduces a specialized temporal index structure for temporal RDF, which significantly improves retrieval performance of some types of temporal queries. The paper closes with an empirical evaluation of the introduced techniques and a discussion of the limitations and future work.

5.2 Related Approaches

The introduction of time into Semantic Web Data structures is not a novel challenge. In this section we will discuss some of these propositions and highlight their strengths and weaknesses. Specifically, we will discuss temporal extensions to OWL and Description Logic,

methods for encoding time in the data model, and maintenance of graph versions.

5.2.1 Temporal extensions

Different approaches cope with the temporal extension of Semantic Web or Semantic Web related technologies. For example, Artale and Franconi [Artale and Franconi, 2005] propose to extend Description Logics with temporal features, while Welty and Fikes [Welty and Fikes, 2006], Kim et al. [Kim et al., 2008] and the tOWL [Milea et al., 2008] project aim at introducing temporal entities into OWL. O'Connor et al.[O'Connor et al., 2007] investigated the querying of OWL data from the medical domain. In their approach, unlike ours, they use SWRL rules to query the KB. Our goal is to embed time into RDF, i.e., lower Semantic Web layers. We, therefore, base our approach on the foundations introduced by Gutierrez et. al. [Gutierrez et al., 2007, Gutierrez et al., 2005]. Gutierrez and colleagues also present a formal sketch for querying temporally enhanced RDF-graphs but did not propose a syntax to be used in, e.g., SPARQL. Even though temporal query languages have been extensively researched in the database community and lead to the TSQL2 query language [Snodgrass, 1995], we were not able to find any analogous extension of the RDF query language SPARQL.

5.2.2 Time encoded in the data model

A commonly used alternative to encoding time as a semantic element in the representation is to express temporal restrictions and validities, time is modeled as a part of the user's data model. Usually, a datatype property is defined with a range to one of xsd's date datatypes [Campbell et al.,]. Pursuing this approach, two different

problems arise:
First, we *loose the semantics of time*. Only a human understands that, e.g., the `hasBirthday` property denotes the start of the validity of an instance of the `Human` concept and that the completely different property `manufacturingDate` of the concept `Car` bears semantically comparable information.
Second, *temporal properties can only be attached to concepts or instances*. Whenever a relation needs to be annotated with temporal validity information, work-around solutions such as relationship-reification need to be introduced. As an example, consider the property `isMemberOf`, which is valid in a certain time interval. One work-around would be the reification of the property using a "Membership" blank-node with the properties `start`, `end`, `memberOf`, and `member`. This is cumbersome and leads to an increased query complexity requiring the inclusion of the blank-node in the query's graph pattern.

5.2.3 Graph versions and version management systems

Another approach to model time is to maintain multiple versions or temporal snapshots of the graph. Indeed version management systems are very popular to track the evolution of files (such as program source code) and their content. *SemVersion* [Völkel et al., 2005] is such a version management system with a focus on ontology evolution management based on named graphs. However, *SemVersion* uses its own data model that contains the user's ontology and its evolution. Our proposition is not to introduce a new data model, but, storing the elements of the users' data model inside different named graphs according to their temporal context.
The advantage of versioning graphs is that the data model (usually) stays untouched. The major disadvantages are that they (1) limit the

temporal semantics to certain time-points (i.e., snap-shots at given points in time) making the use of even slightly more involved temporal semantics, such as interval queries, impossible, (2) do not expose time explicitly but implicitly through snapshots making temporal reasoning complicated, and (3) will force users to access many snapshots if an interval of interest spans many versions forcing them to incur a potentially huge overhead due to redundant information in multiple versions. Our proposed solution will avoid all of these disadvantages.

5.3 A Temporal Syntax for RDF

In this section we introduce our temporal representations of time. Specifically, we introduce two different representations. The first is for internal use only and facilitates the addressing and indexing of temporal entities (e.g. for a triple store). The second provides the basis for a machine processable exchange format considering the semantics of time.

5.3.1 Internal representation

As in most of the approaches dealing with temporal entities, we model time as a 1-dimensional discrete value (i.e. no branching time). More formally, each valid time point is defined as $\tau \in \mathbb{N}_0$. A time interval consists of two time points s (for start) and e (for end) such that $s, e \in \tau | s \leq e$. By allowing s to be equals e, we are able to express time points as intervals. Consequently, we will henceforth only use the interval notation. Note that this integer representation is an abstracted, simplified form of time which looses the exact semantics of time (e.g., whether it is a year number or version

number). For most *internal* representations this is sufficient, since we only need to operate on the relative relations of values, i.e., $>$, $=$ and $<$. Furthermore, we allow half-bounded and unbounded intervals. We employ three special intervals that denote time periods which are open in at least one direction: $[0, e]$, $[s, \infty]$, and $[0, \infty]$. Since we restricted the range of valid time points to $\mathbb{N}_0$, 0 denotes the lowest possible value (i.e., the beginning of time). There are no time points before 0. The maximum time value, different from what ∞ might imply, does not lie ahead. It is allocated with the actual time during execution.

5.3.2 Exposed representation

As a machine and human legible exchange format an exposed, semantic representation / interpretation is necessary, which may consist of the well-known xsd datatypes. In this approach we use the OWL-Time [Hobbs and Pan, 2005] ontology that defines `Time-points`, `Intervals` and different date formats. However, we extended OWL-Time with a new date format type called `Integer-Time` to express non-calendaric time representations such as version numbers.

Analogous to the internal and conceptual schema of a DBMS the mapping of the internal to the exposed representation and vice-versa is a storage system dependent strategic decision that impacts the storage and retrieval performance. To map the above mentioned borderline values 0 and ∞ to semantically more expressive external representations, we use the literal values EVER and NOW. Hence, the internal $[0, \infty]$ interval is mapped to its semantic counterpart [EVER,NOW] which refers to a time interval whose elements have always been valid. Any data set without temporal information will be considered to be valid in this “eternal” interval.

5.3.3 Storage Format and Syntax

As mentioned before, we use named graphs [Carroll et al., 2005] to enable the temporal data representation. We refer to a set of temporally related named graphs as a temporal graph. Each time interval is represented by exactly one named graph, where all triples belonging to this graph share the same validity period. To add an element to a temporal graph, the algorithm shown in Listing 5.1 is executed.

```
1  # retrieve validity interval
2  [start, end] = tripleToAdd.validity
3  if tripleToAdd containsOneOrMore blankNode
4          # generate URI for blank-node
5          tripleToAdd = convertBlankToURI(tripleToAdd)
6  endif
7  # if no named graph for the interval exists then make one
8  if !namedGraph[start, end].exists
9    create new namedGraph[start, end]
10   # add the new named graph to directory in default graph
11   namedGraph.defaultGraph += namedGraph[start,end].temporalInfo
12 endif
13 namedGraph[start,end] +=  tripleToAdd
```

Listing 5.1: Pseudo Code for an insert operation into a temporal graph

As the listing shows, we first ascertain the triple's temporal validity. Note that the extraction of the validity is dependant on the way time is represented in a data set. Then, we check, if it contains a blank node. Blank nodes are especially problematic as their validity is restricted to the node's parent graph. In our case, however, where the collection of the interval-representing named graphs logically form the overall graph, it might make sense to have blank-nodes that span multiple intervals and, thus, multiple named graphs. This requirement is akin to the implementation employed by the Intellidimension RDF Gateway's quad approach [Carroll et al., 2005], which deviates from the *W3C* recommendation in that respect. To

adhere to the W3C recommendation we simply assign a URI to any blank node circumventing the blank node's spirit but fulfilling our premise of using off-the-shelf tools – in our opinion an acceptable compromise between adhering to the spirit of RDF and practicality. Next, we establish, if a named graph representing the interval already exists. If not, then we generate a new named graph for this interval and add information about it to the named graph directory in the default graph. Finally, we add the triple to the appropriate named graph. Note that we do not allow the same triple to appear in multiple temporal contexts. Therefore, a temporal RDF framework needs to take care that a triple only exists once, but maybe, due to an update of the temporal information travels from one named graph to another.

As apparent from this procedure we use the default graph as a directory or container for the semantic information about the intervals. Since every interval is uniquely identified by the URI of its named graph, vital as well as additional information can be provided in the default graph. Vital information includes the start and end time of an interval in any time format allowed by OWL-Time. Additional information includes relations between intervals such as `after`, `before`, or `overlaps`. Since the additional information about intervals can be inferred from the start and end time of an interval, a temporal reasoning system could entail these relations.

With this approach, we can benefit from well-established, scalable support of named graphs in storage systems without having to implement any adaption to temporal RDF. As mentioned above, temporal validity information could also be provided using reification. However, to express the same amount of information, reification would consume 15 times more triples than named graphs seri-

ously questioning the scalability both in terms of storage space and, more importantly, in terms of retrieval run-time.

5.4 The τ-SPARQL Query Language

Having established a storage format in Section 5.3.3, in this section we focus on the retrieval of these triples. It introduces τ-SPARQL, an extended syntax for SPARQL to express temporal queries. We introduce τ-SPARQL by discussing its two major usage formats: time point Queries and temporal queries.

5.4.1 Time Point Queries

Time point queries aim at retrieving information valid at a specified point in time. This is of special importance whenever a snapshot in the past needs to be retrieved from a dataset. Similar to the `FROM` statement in SPARQL to select a dataset (or graph), we define a `FROM SNAPSHOT` τ expression to select a specific point τ in time. The `FROM SNAPSHOT` τ expression signals the query engine to evaluate the query's graph pattern only on graphs-elements valid at the time point τ, where τ has to be a literal time value (the literal type is depending on the underlying time format). The query in Listing 5.2, for example, retrieves all `foaf:Person`s that were valid (i.e. alive) in 1995.

```
1 SELECT ?person FROM SNAPSHOT 1995 WHERE{
2         ?person a foaf:Person .
3 }
```

Listing 5.2: τ-SPARQL: Time point query

5.4.2 Temporal Queries

Complementing time point queries that query the graph valid at a given point in time, temporal queries allow the usage of wild card intervals and time points. These wild cards can be used to bind a variable to the validity period of a triple or to express temporal relationships between between intervals. τ-SPARQL allows one form of temporal wildcards, `[?s,?e]`, which binds the literal start and end values. Note that whilst τ-SPARQL explicitly binds the two variables `?s` and `?e` that can be used elsewhere in the query, a third implicit variable `?g` is internally bound to the URI of the named graph that represents the interval defined by *[?s,?e]*. Hence, the expression used in the interval context *[?s,?e]* is bound to the named graph URI while the partial variables *?s* and *?e* are standard SPARQL variables which are bound to the literal start and end value of the interval's validity. Therefore, `?s` and `?e` can occur in any part of the query specified by the SPARQL syntax, e.g. in `FILTER` or `ORDER BY` expressions. In Listing 5.3, an example query is shown, which retrieves a person's URI and the start of the validity period (i.e. the person's birthday). Since a temporal annotation is based on triple level, the interval variables (or specifications) could be defined before or after each subject, predicate, or object. For the sake of consistency, we define that a temporally annotated triple pattern (i.e., quad pattern) must always start with the temporal variable part.

```
1 SELECT ?s ?person WHERE{
2         [?s,?e] ?person a foaf:Person .
3 }
```

Listing 5.3: τ-SPARQL: Selection of validity period

To query for relations between intervals, two interval wildcards can be connected with the properties defined by the OWL-Time ontology. These properties are the well-know Allen Interval Relations [Allen, 1983]. Listing 5.4 shows a query to retrieve all the `foaf:Person`s that could have known Albert Einstein because their lifespan overlaps with Einstein's. In literature, this kind of patterns are referred to as *temporal join* [Snodgrass, 1995].

```
1 <PREFIX time: http://www.w3.org/2006/time#>
2 <PREFIX foaf: http://xmlns.com/foaf/0.1/#>
3 SELECT  ?s2 ?e2 ?person WHERE{
4         [?s1,?e1] ?einstein foaf:name "Albert Einstein" .
5         [?s2,?e2] time:intervalOverlaps [?s1,?e1] .
6         [?s2,?e2] ?person a foaf:Person .
7 }
```

Listing 5.4: τ-SPARQL: Interval relations

5.5 Mapping τ-SPARQL to Standard SPARQL

In this section we show, that the τ-SPARQL syntax extension can be rewritten without any loss of expressivity to standard SPARQL. Therefore, τ-SPARQL offers an intuitive syntax for the composition of queries (for human consumption), while the rewritten form can be evaluated as standard SPARQL using any query engine supporting named graphs using their built-in standard query optimization and storage formats scalability.

5.5.1 Mapping Time Point Queries

Rewriting a time point query requires three steps: First, the respective intervals for a time point needs to be determined. Second, the

elements belonging to these intervals need to be combined to a new, virtual data set. Finally, the graph pattern needs to be matched against this new data set.

As described above, the information about the validity of an interval is encoded in the default graph of the named graph set. To extract the intervals valid at a given time point, the partial query shown in Listing 5.5 has to be added to the initial query in the `WHERE` clause.

```
1 ?g time:hasBeginning ?start .
2 ?start time:[inInteger|inXsdDateTime|...] ?s .
3 FILTER(?s <= <TP> || ?start = NOW) .
4 ?g time:hasEnd ?end .
5 ?end time:[inInteger|inXsdDateTime|...] ?e .
6 FILTER(?e >= <TP> || ?end = EVER) .
```

Listing 5.5: Partial query to determine valid intervals at a given time point

The pattern in Listing 5.5 will bind variable `?g` to all the named graphs containing triples valid in time point `<TP>`. The expression `[inInteger | inXsdDateTime | ...]` is evaluated according to the literal type of `<TP>`. If it is a `xsd:Integer`, then the `time:inInteger` property is selected. Analogously, a `xsd:dateTime` will be represented by the `inXsdDateTime` property. Note that both the τ-SPARQL query language and the representation format are open to additional time formats as long as the rewriting algorithm is aware of the acceptable correct formats and their respective mappings.

In a last step of the rewriting process, the initial graph pattern of the query needs to be restricted to the intervals bound in `?g`. SPARQL offers this functionality with the `GRAPH` keyword. This is done by extracting the pattern inside the `WHERE` clause of the query and inserting it enclosed by `GRAPH ?g {<pattern>}`.

5.5.2 Mapping Temporal Queries

To map the queries that select the validity of a triple pattern and/or relations between validities, the rewriting process is slightly more complex. We define that all classic SPARQL variables `?v` belong to a set C. The intervals `[?s,?e]` to belong to the set of intervals I, and `?s` and `?e` to belong to the set of point variables P. For each $i \in I$ there exists exactly one URI g belonging to the set of URIs of named graphs G and a pair of time points `?s` and `?e`, which mark its start and end. Any pair of time points `?s` and `?e` can be mapped to an interval i, which in turn can be mapped to a URI from G. Consequently, a quad pattern in the form `[?s,?e] <triple pattern>` can be rewritten as `GRAPH ?x{<triple pattern>}` where `?x` is the URI $\in G$ that corresponds to the interval defined by `[?s,?e]`. Note, that a new variable "`?x`" needs to be chosen for every rewrite. Please note that temporal variables are not allowed in the predicate part.
It is important that whenever a temporal variable is used outside the interval context (i.e., outside a `GRAPH ?x{<triple pattern>}`) then it needs to be treated like a time point statement. Hence, the partial query shown in Listing 5.5 needs to be added to the rewritten τ-SPARQL query.

As an example, Listing 5.6 shows the rewritten standard SPARQL syntax of the query in Listing 5.4.

```
<PREFIX time: http://www.w3.org/2006/time#>
<PREFIX foaf: http://xmlns.com/foaf/0.1/#>
SELECT  ?s2 ?e2 ?person WHERE{
        GRAPH g1 { ?einstein foaf:name "Albert Einstein" .}
        ?g2 time:intervalOverlaps ?g1 .
        GRAPH g2 { ?person a foaf:Person }.
        ?g2 time:hasBeginning ?start .
        ?start time:inInteger ?s2 .
        ?g2 time:hasEnd ?end .
        ?end time:inInteger ?e2 .
}
```

Listing 5.6: Rewritten τ-SPARQL query

5.6 An Index Structure for Time Intervals

In the previous section we demonstrated how to rewrite τ-SPARQL time point queries to standard SPARQL. However, the performance of time point queries can be raised by providing a dedicated index structure. Temporal index structures have been discussed extensively in the database community. In a recent publication, [Pugliese et al., 2008] proposes the tGRIN index structure for temporal RDF w.r.t. the temporal as well as the structural neighborhood of triples. Our requirement, however, is the retrieval of intervals valid in a certain time instant. Therefore, we propose such an index structure that supports the efficient retrieval of intervals valid in certain time instants. This is not a base index of triples, it is rather a meta-index containing named graphs which, by themselves, have indices of their triples.

Specifically, we use a tree-based index structure which stores the validity start and end values in an ordered manner. To achieve a fast retrieval of all valid intervals at a specified time point we would need to store each interval multiple times into this index: in each

time point of its validity. E.g., an interval [0,100] would appear 100 times in the index. This is an unpractical approach as it requires a large amount of space. Similar to the mpeg [Pan, 1995] compression algorithm we apply the concept of key frames (more precisely: I frames) to our interval index structure. Similar to a key frame, a key index element stores all the intervals valid at this time point. Between the key indices only the deltas are stored. We refer to this index structure as keyTree index. This index structure is very similar to the Time Index proposed by [Elmasri et al., 1990]. The Time index maintains incremental buckets which refer to the deltas between two index elements. However, unlike our keyTree, the Time Index does not allow to select a key index distance, it rather selects the key indexing points whenever an interval starts or ends. This can lead, in datasets with very dense intervals, to oversized indices. Our index offers the parameter *key index distance* which permits a storage system to adapt to different temporal kinds of datasets, e.g., whether the time intervals distribution is dense or sparse. The retrieval strategy of the keyTree index can be summarized as follows:

- if the asked time instant coincides with a key index, return all elements in the key index

- if the asked time instant does not coincide with a key index, go back to the immediately preceding key index. From there, walk forward and apply all the deltas until the asked index element is reached and return the set with the applied deltas.

We will show an evaluation of this index structure in the next section with a comparison to an alternative index.

Data Source	Time Model	Data Period	# Triples	# Intervals
EvoOnt	Versioned graphs	2001 - 2007	22 millions	2505
Parliamentarians	Encoded in data model	1848 - 2008	50'000	38'182

Table 5.1: Datasets

5.7 Evaluation

For the evaluation of our approach we are using two different data sources. We used the EvoOnt data [Kiefer et al., 2007b, Kiefer et al., 2007c] that consists of software source code information. Specifically, we used information about the source code of the compare plugin of the Eclipse Project (*org.eclipse.compare*). 306 releases created in the period between 2001 and 2007 were available. The second data source is a data set describing Swiss parliamentarians (with information about year of birth and death as well as start and end of service periods) between 1848 and 2008. The latter data has to be seen, for this approach, as a worst case, as almost every triple is defined in its own time interval. Table 5.1 lists the characteristics of the data sources.

All evaluations in this section were computed on in-memory graphs using the ng4j [Bizer et al., 2005] named graph API. To ensure that we compare different approaches rather than the quality of the implementation of the RDF APIs, we also load the comparison data sets into one single named graph and run queries against the ng4j implementation (which is based on Jena [Carroll et al., 2004]).

All the evaluations were executed on an Intel Core2 Duo 1.6 GHz system with 4 GB of RAM running Windows Vista.

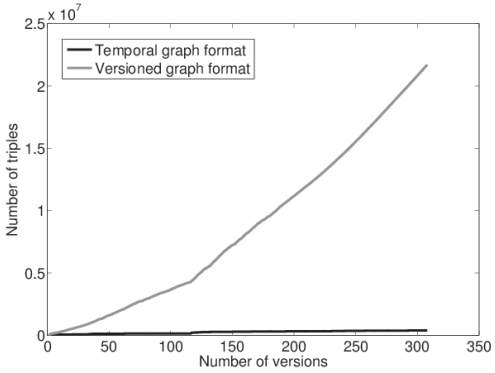
5.7.1 Dataset Conversion

Our first evaluation covers the conversion from the respective time representation format into temporal RDF according to the method presented in this work. First, we converted the EvoOnt data set into a temporal graph. Figure 5.1(a) shows the vast reduction of triples in the temporal graph approach due to the reduction of redundancies. Having a graph that contains the information about all the 306 versions, the number of triples reduce from 22 millions in the versioned graph approach to 385'000 in the temporal graph. The reason for this huge reduction is depicted in Figure 5.1(b). The fraction of triples changing between versions of the graph (i.e. between the software versions) is usually very small (far below 5%). As a consequence, 95% of the graph (i.e., the unchanged triples) are stored redundantly.

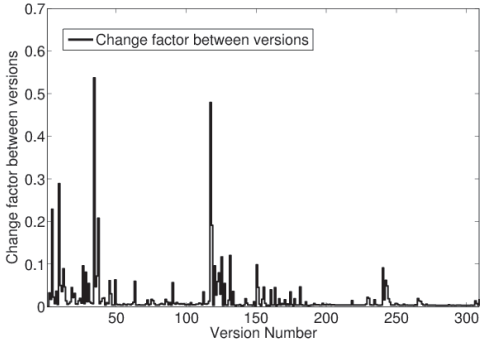
5.7.2 keyTree Index

To evaluate the efficiency of our keyTree index structure introduced in Section 5.6, we compared our index structure to an in-memory ordered list (Java's TreeList implementation) that contains the intervals' start and end times. Additionally, we added three special sets of intervals containing the borderline values [EVER,x], [x,NOW] and [EVER,NOW] to this baseline implementation.

We evaluated two different scenarios on our datasets. The first is the build up time for the indices with different values for the keyTree index' parameter *key index distance* (i.e. the number of time instants between two key indices). Secondly, we measured the retrieval time



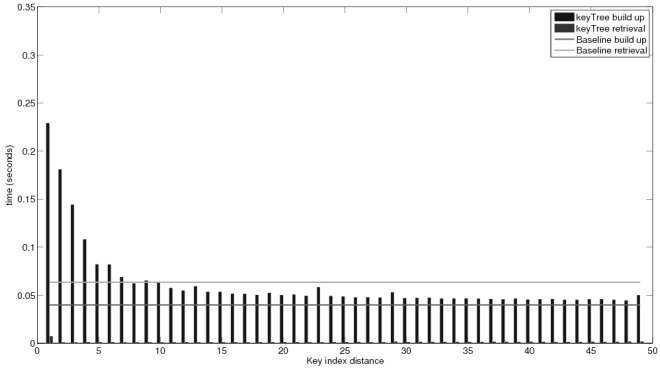
(a) Growth of number of triples. Gray line: versioned graph, black line: temporal graph.



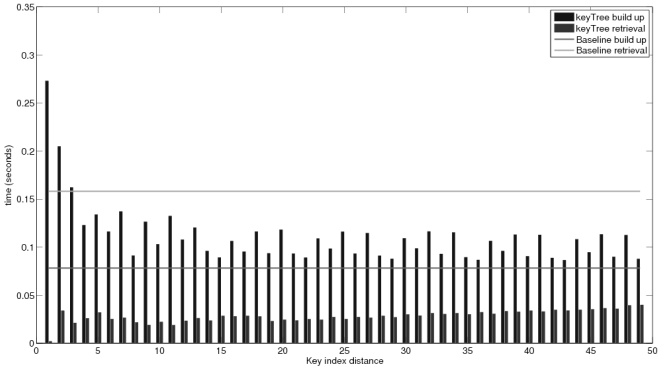
(b) Change factor between the different versions in the dataset.

Figure 5.1: Conversion from graph versions to a temporal graph

for the set of valid intervals in a certain time instant. The following figures show the build and retrieval times for the Parliamentarians (Figure 5.2a) and EvoOnt (Figure 5.2b) datasets. All values were calculated by running each operation 100 times and plotting the average time of these runs.



(a) keyTree index build up and retrieval of the Parliamentarians dataset



(b) keyTree index build up and retrieval of the EvoOnt dataset

The figures show, that the keyTree index structure never outperforms the baseline implementation in terms of build time. However, when amortizing the build time over multiple retrievals the use of the more sophisticated keyTree seems to pay off. As a conclusion, the keyTree index is at least three times faster than a "standard" one. The reason for the decreased performance of the baseline is that for each retrieval request the whole index needs to be traversed to check whether there are elements that are valid during the asked time point.

5.7.3 Timepoint Queries

Next, we run time point queries using both, the Parliamentarians and EvoOnt data sets. Since a time point in the EvoOnt dataset is represented by a snapshot graph, we equate the time to load the graph into memory with the querying time. We also compare the querying time for the temporal graph with the τ-SPARQL triple pattern approach to determine the valid intervals. The evaluated query consisted of retrieving all the triples in a defined time instant (Parliamentarians dataset: 1995, EvoOnt: 5). Additionally we queried the datasets using our keyTree index. Figure 5.2 shows the execution times of the queries in the different setups and datasets.

Again, the versioned graph approach does not allow to run time point queries, instead, we listed the time needed to load the graph into memory which is the equivalent task needed to select the valid triples in a specific time instant. The results shown in Figure 5.2 reflect the different nature of the datasets. The Parliamentarian data consists of many different time intervals with relatively few triples in each interval whereas the EvoOnt dataset has less intervals but many more triples per interval. On this note, the keyTree index structure can greatly reduce the query execution time in the EvoOnt

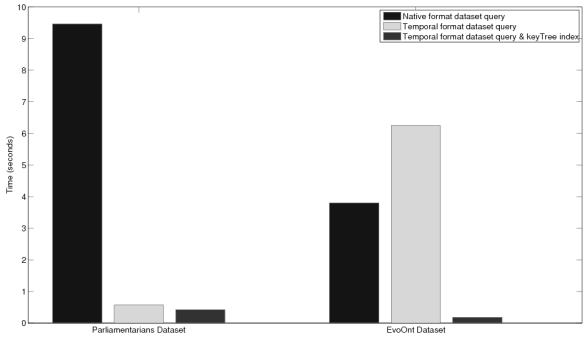


Figure 5.2: Execution times of time point queries using different datasets and query strategies.

dataset because the query engine encounters a vastly reduced number of triples to run the pattern against. The long query execution time for the non-temporal query on the Parliamentarians dataset is mostly due to the complex query pattern using UNIONs to retrieve all elements valid at the desired time point.

5.7.4 Temporal Queries

In a final set of evaluations we cover temporal queries. Since such queries (e.g. after, overlaps) are not possible to run against a dataset that uses versioned graphs, we only used the Parliamentarians dataset. The query we evaluated was to retrieve all the parliamentarians complete service periods while "Kurt Furgler" was alive (1924-2008). The τ-SPARQL query uses the `time:intervalDuring` relation, while the classical SPARQL comparison query checks every service period whether it is enclosed by the life time of "Kurt Furgler". Figure 5.3 shows the execution times of the two queries.

Again, it can be seen, that the temporal approach is performing (almost 50%) better because of the reduced complexity of the query pattern. However, the execution time increases whenever all the interval relations are entailed and written into the default graph. For the query, we only entailed the interval relations asked by the query and omitted additional ones. But, as mentioned above, this can be absorbed by using a temporal reasoner that provides the interval relations without materializing them into triples.

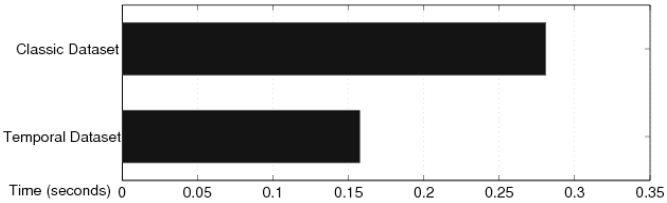


Figure 5.3: Execution times of temporal queries using temporal and non-temporal query strategies.

5.8 Conclusion, Limitations and Future Work

We presented an applied temporal RDF using named graphs. In detail, we presented a syntax as storage format which allows the annotation of RDF triples with temporal validity intervals. This approach is as minimally invasive as possible to preexisting storage and querying systems. Furthermore, we introduced T-SPARQL, an extension to express temporal queries. We showed, that τ-SPARQL can be directly mapped to standard SPARQL. Additionally we presented a specific index structure for temporal intervals which improves the retrieval time of time point queries. Finally we evaluated our approach comparing our approach to classical graph ver-

sioning and time representation and could show that our temporal RDF and τ-SPARQL approach enables queries that are either impossible (versioned graph) or only achievable with very complex query patterns (time in data model) resulting in increased performance in terms of answering time of queries. We showed cases where our approach outperforms existing ones and circumstances where our approach does not completely outperform existing ones. However, we showed, that a combination with our keyIndex structure can overcome this lack of performance. Moreover, when using temporal RDF, the total number of triples can be largely reduced by removing redundancies which releases resources in each of the storage, reasoning and querying systems. This has a positive effect on disk space usage and response time.

As a limitation we have to mention the lacking availability of temporal reasoners. When materializing the interval relations such as *overlaps*, *meets*, or *before*, the number of temporal relations grows exponentially which makes the materialization in triple form unpractical. Furthermore, our approach performs best on data that has multiple data elements valid within the same interval. If each triple's validity is distinct then one named graph needs to be defined for each triple, which can lead to a decreased performance. However, we believe that many application scenarios exist, where such an extreme interval mapping is not the case.

Our future work will have to concentrate at addressing the current limitations. One possibility is to explore the the entailment of new temporal relations based on our keyTree index. Additionally, we plan to further evaluate our approach with more complex query patterns from different domains of application.

6

Hτ-Index: Using Temporal Index Structures to Efficiently Entail the Allen Calculus

Jonas Tappolet and Abraham Bernstein

Reviewed (revise and resubmit) by the Journal of Web Semantics

ABSTRACT One oftentimes overlooked element of Semantic Web data is the fact that it changes over time. Indeed, most information, such as affiliation of players to sports teams or a person's job change frequently. Hence, the Semantic Web should take a temporal or at least a versioned view on supplied data. Whilst formalisms for the representation of temporal and/or versioned data have been proposed the efficient processing of these data is hampered by the

scarcity of time and space efficient storage approaches. This paper presents the Hτ-Index, an index structure that efficiently maps time intervals to a B+-tree using the Hilbert space-filling curve. A major characteristic of the presented index structure is the direct entailment of the 13 Allen interval relations (semantics of time) – to our knowledge the first index with this crucial property. In addition, we introduce a decision procedure that further improves the retrieval performance of the Hτ-Index.

To evaluate the efficiency of the Hτ-Index and its extension we empirically compare it against two of the currently best performing temporal indices called UB-Tree and MAP21. Using a cost function that weights index scans and seeks we show that the Hτ-Index outperforms the UB-Tree and MAP21 by more than 50% (in terms of retrieval costs).

6.1 Introduction

Temporal RDF has been proposed for a variety of applications [Tappolet and Bernstein, 2009, Koubarakis and Kyzirakos, 2010]. As an example consider a knowledge base that keeps track of information about sports teams and its players. It would be disadvantageous if the old team memberships would be overwritten with the information about a player that newly joined the team. Tagging the information with a time period indicating their validity would alleviate this problem. Note that the validity should be modeled as a distinct type. Else, for example when modeling time-specifying elements as strings, temporal semantics (such as was event A before event B, or did it happen at the same time) would get lost and could only be retrieved using special, format-aware programs. Consequently, the

Semantic Web needs to become time-aware to provide the foundation of long-living knowledge bases.

Whilst a number of temporal extensions to RDF have been proposed [Gutierrez et al., 2007, Milea et al., 2008] ensuring a correct representation of these kinds of data the question of how to query temporal data efficiently is still open. The query "in what sports teams a player played and who his teammates where during his tenures" (in the different teams), for example, requires a temporal join: first the temporal information about the team memberships of the player need to be retrieved and then the corresponding time-intervals need to be intersected with the temporal information of potential teammates. Such temporal joins are very expensive operations because they require, in the worst case, testing for overlaps with every other time interval in the knowledge base.

To aggravate the problem testing for *overlaps* is only one type of temporal query. Allen [Allen, 1983] identified 13 different types of relations, which form the complete set of interval relations. Besides overlapping, time intervals can *meet*, *start*, *finish*, and *contain* each other. All of those relations have an inverse counterpart. Additionally, intervals can be *before / after* each other and, finally, they can be *equal*. The entailment of these symbolic relations (temporal semantics) from the numerical start and end time is a challenging task as it includes a multitude of testing steps.

Most practical query environments retrieve data from an on-disk representation. Hence, all interval information needs to be loaded into memory where the comparisons can be executed. Retrieval from (both spinning- and flash-) disk is still the most time-consuming operation in a computer system. Additionally, temporal knowledge bases have the habit of constantly growing over time (as updates will not overwrite the old values). Indisputably, such a

system will eventually reach a number of time intervals where the processing time for a query renders the whole system unusable.

In this work, we propose the novel Hτ-Index—*an index structure that stores time intervals according to their semantic relationships* (*i.e.* the Allen interval relations). By projecting time intervals to the parameter space of a fractal function, each Allen interval relation can be mapped to a region in the index space. The result is an on-disk index that can directly interpret temporal semantics only retrieving data that adheres to temporal constraints as defined by a query. This avoids moving irrelevant information through the disk-memory bottle-neck for temporal reasoning speeding up the query process considerably.

Specifically, the Hτ-Index maps each element into a 2-dimensional space, where the dimensions represent the start and end time of the validity of a given triple. The main challenge is to take this 2-dimensional space and store it on the linear structure of current-day hard-drives. Current storage-hardware essentially support two operations: firstly, for the seek operation an internal pointer can be set to a storage address (in a spinning-disk, this relates to the positioning of the read-head to a physical position). Secondly, for the scan operation the data is be retrieved sequentially from the current position. Note that seeks are always more expensive than scans (interestingly also in modern-day flash drives although not as pronounced as with spinning disks and depending on the implementation of the disk access layer). Hence, seeks should be reduced to a minimum. Using the Hilbert space-filling curve the Hτ-Index maps the 2-dimensional time space to a single dimension, which in turn can be indexed by a disk-optimal B+Tree. Given that almost every database implements a B+Tree our method can therefore be added to most standard database systems without touching its implementa-

tion. For querying, we identify the relevant temporal interval in the 2-dimensional space, map this to a key-space via the Hilbert function, and then use the B+Tree to map it to seek and scan operations. Once the query area is identified in the 2-dimensional space, this area can be mapped to a set of seek and scan operations. Comparing our approach to the only other index with comparable characteristics, the UB-Tree [Bayer, 1997], we show that our approach is superior and outperforms the UB-Tree by more than 50%. Additionally we compare our approach to the MAP21 [Nascimento and Dunham, 1999] approach (which showed a similar improvement).

Hence, the contributions of this paper are (1) the mapping of Allan relations to two-dimensional space, (2) the presentation of a deterministic method for efficiently retrieving rectangular shapes using a Hilbert-curve, and (3) the combination of all these techniques to a Hτ-Index that directly interprets Allen relations optimizing temporal retrieval.

In the next section we lay the foundation by presenting the related work. Section 6.3 explores approaches for mapping time to two-dimensional space, which is followed by a discussion of how to use space-filling curves to further map the two-dimensional time-structures to a one-dimensional key-space(Section 6.4). In Section 6.5 we discuss how the resulting index structures can be queried and compares the different approaches to each other. We close with a discussion of the limitations and future work.

6.2 Related Work

In this section we will briefly review related approaches to our work. We have divided the section into two parts. Firstly we will review sophisticated indices for time intervals. Secondly, we will present related multi-dimensional index approaches, because the indexing of time intervals can be mapped to a 2-dimensional instance of a universal n-dimensional index.

6.2.1 Time Interval Index Structures

One of the first indices that explicitely addressed time intervals was the Time Index by Elmasri *et al.* [Elmasri et al., 1990]. Along the time line, so-called time buckets are created. Every time bucket stores the complete set of intervals valid at this specific instant. Between the buckets, only the deltas are stored. It is obvious, that this index is optimized for intersection queries which can be retrieved with relatively few index lookups. A downside of the time index is its high storage overhead (especially for long lasting intervals) since an interval can ocupy many different buckets consuming storage.

The TP-Index [Shen et al., 1994] is a 2-dimensional index structure that uses a start-length mapping (see Section 3). The 2-dimensional space is divided into areas with different shapes (*e.g.* rhomboids). The sub-areas are then respectively split into sub-sub-areas. Following this procedure, a tree of area elements can be built and stored on disk. The querying consists of the selection of the query area and the retrieval of this area from the tree structure. The authors of the TP-Index also presented a mapping of the start-length mapping onto a standard B-tree. They propose the utilization of 3 B-trees: one traversing the index space horizontally (sweep line), a second one for the vertical traversal as well as a third B-tree contain-

ing the diagonal traversal. In this work, we use a different mapping that makes the retrieval of the semantic relations of time intervals possible. Furthermore, by using the area-preserving Hilbert curve we can omit the space overhead by only using one B+tree where the index space is mapped on. Another index structure using geometric decomposition is the TD-Tree was recently presented and also presents a geometric decomposition method based on triangles which is used for the indexing of time intervals [Stantic et al., 2010].

Pugliese *et al.* [Pugliese et al., 2008] presented tGRIN, a compound index structure for temporal RDF. tGRIN builds on the assumption that query results contain elements that have a high temporal and structural proximity to each other. Those neighboring elements are stored on the same or at least a near page of the tGRIN index. In this work, we are focusing on the semantics of time (*i.e.* the entailment of the Allen interval relations). This is a feature not supported in tGRIN. We will therefore not compare our approach to tGRIN, as both index structures cover different application areas.

6.2.2 Multidimensional Index Structures

Many different multi-dimensional index structures have been presented in the past. In this subsection we will review the two most related to our approach. For a general overview of multidimensional access methods we refer the reader to Gaede and Günther [Gaede and Günther, 1998]. The R-Tree [Guttman, 1984] has been proposed as an index structure for spatial objects. The tree nodes contain the bounding box of the area covered by the elements of the child node. Inserting a new element requires the traversal from the root of the tree. At each node it is checked whether the new element overlaps with the bounding box of the current node. If so, the intersecting branch is traversed until a leaf node is reached. When a node is full,

it is split up, otherwise the element is inserted in this leaf node. The performance of the R-Tree is highly dependent on the efficient organization of the elements within a node.

To that end, the Hilbert R-Tree [Kamel and Faloutsos, 1994] has been proposed which uses the Hilbert order [Hilbert, 1891] to internally organize the elements. The advantage of Hilbert numbers is that if they have a low difference to each other, they also represent points in the parameter space that are near neighbors. In our work, we propose the usage of standard B+trees instead of a sophisticated index structure and use the Hilbert order to project the 2-dimensional data points onto a B+tree which is available in almost every database system.

Retrieval algorithms have been proposed for rectangle retrieval from the Hilbert order by Jagadish [Jagadish, 1990] and Lawder [Lawder, 2001] for the purpose of high-dimensional indexing. We base our work on the recently proposed algorithm by Chen *et al.* [Chen et al., 2007] for the calculation of the Hilbert index number and propose a novel approach for the optimized retrieval of rectangles based on a cost function.

Another index structure is the UB-Tree [Bayer, 1997, Tropf and Herzog, 1981]. The general idea is as well the projection of a multi-dimensional index space onto a 1-dimensional B-tree. Bayer proposes the usage of the Z order to reduce the dimensions. This is a valid approach since the calculation is straightforward using bitwise shifting and therefore has a negligible computational complexity. It has always been stated that the Hilbert order has superior locality characteristics, nevertheless, its recursive calculation is cost intensive and error prone [Goldschlager, 1981]. Due to the recent publication of an iterative calculation algorithm for the Hilbert order [Chen et al., 2007] the computational complexity can be highly reduced. In

our experiments, we will compare the Hilbert order directly with the Z order and quantify the expression "better locality characteristics" with numbers that relate to the number of index jumps (seeks) and reading operations (scans).

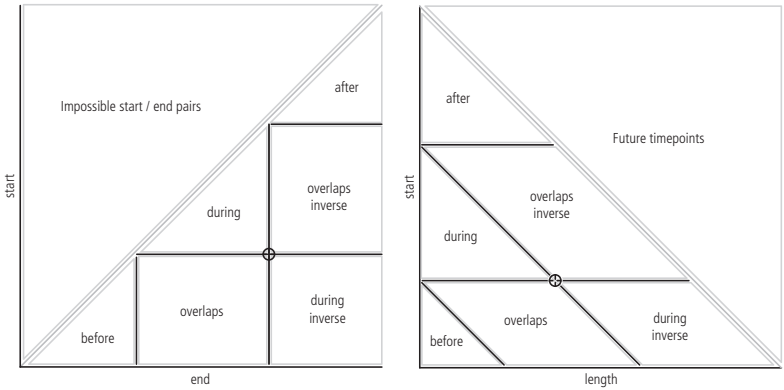
In the following section we will present our mapping approach and its advantage for the preservation of the semantics of time.

6.3 Mapping Time intervals into 2d-Space

In this section we present time interval mappings to a 2-dimensional space. Further, we lay out the idea of the identification of Allen interval relations in this index space.

Time intervals are defined by the two explicit values *start* and *end* ($i_{[start,end]}$ with $start \leq end$) . A third, implicit element is *length* (=*end-start*). When mapping time intervals to a 2-dimensional space, two different mapping types can be applied. Either, end and start are directly mapped to the x and y axis, or, as an alternative mapping the duration of an interval and its starting point can be mapped to the x and y axes (note that alternative mappings such as start-end are rotational or mirrored representations of the two basic mappings). Those two mapping types have different properties when identifying the Allen-Interval areas. Figures 6.1a and 6.1b show these areas for the two mapping types. The size and position of the Allen-areas are always relative to a specific time interval which has to be provided (circle in Figures 6.1a and 6.1b).

The major difference between the mappings presented in Figures 6.1a and 6.1b are the different predominant shape types for the Allen-areas. For the length-start mapping, Allen-areas are rhomboids or triangles. For the end-start mapping, on the other hand, the areas are triangular or rectangular. We can even omit the distinc-



(a) The start-end mapping. (b) The start-length mapping.

Figure 6.1: Two mapping types. Note that the marked areas are relative to the selected time interval (in the figure highlighted with a circle).

tion between triangle and rectangle shapes in the end-start mapping. Since the upper half of the index space in the denotes invalid points. There we can extend the triangle shapes to rectangles without any effects on the number of elements in that area. Consequently, we can reduce the problem of the entailment of Allen-interval relations to the retrieval of rectangles from a 2-dimensional space when using the end-start mapping whereas in the start-length mapping we have to retrieve both triangles and rhomboids. This is the rationale behind choosing this mapping type for our Hτ-Index. A more formal definition of the Allen-areas of the end-start mapping is listed in Table 6.1. For every relation type, we list the (x, y) pairs of the Allen-area corner points $(a_{bl}, a_{br}, a_{tl}, a_{tr})$. Those corner points are depending on $p_{x,y}$ which is the point in the index space that relates

to the time interval $[x, y]$. The interval $[x, y]$ is the bounded part of the asked Allen-relation.

	a_{bl}	a_{br}	a_{tl}	a_{tr}
before	$(0, 0)$	$(p_x - 1, 0)$	$(0, p_y - 1)$	$(p_y - 1, p_y - 1)$
overlaps	$(p_y + 1, 0)$	$(p_x - 1, 0)$	$(p_x - 1, p_x - 1)$	$(p_x - 1, p_y - 1)$
overlaps inverse	$(p_x + 1, p_y + 1)$	$(now, p_y + 1)$	$(p_x + 1, p_x + 1)$	$(now, p_x + 1)$
during	$(p_y + 1, p_y + 1)$	$(p_x - 1, p_y + 1)$	$(p_y + 1, p_y - 1)$	$(p_x - 1, p_x - 1)$
during inverse	$(p_x + 1, 0)$	$(now, 0)$	$(p_x + 1, p_y - 1)$	$(now, p_y - 1)$
after	$(p_x + 1, p_x + 1)$	$(p_x + 1, now)$	$(p_x + 1, now)$	(now, now)
finishes	$(p_x, 0)$	$(p_x, 0)$	$(p_x, p_y - 1)$	$(p_x, p_y - 1)$
finishes inverse	$(p_x, p_y + 1)$	$(p_x, p_y + 1)$	(p_x, p_x)	(p_x, p_x)
starts	(p_y, p_y)	(p_y, p_y)	$(p_x - 1, p_y)$	$(p_x - 1, p_y)$
starts inverse	$(p_x + 1, p_y)$	$(p_x + 1, p_y)$	(now, p_y)	(now, p_y)
meets	$(p_y - 1, 0)$	$(p_y - 1, 0)$	$(p_y - 1)$	$(p_y - 1)$
meets inverse	$(p_x + 1, p_x + 1)$	$(p_x + 1, p_x + 1)$	$(p_x + 1, p_x + 1)$	$(p_x + 1, p_x + 1)$
equals	(p_x, p_y)	(p_x, p_y)	(p_x, p_y)	(p_x, p_y)

Table 6.1: The x and y values of the allen areas. $a_{bl}, a_{br}, a_{tl}, a_{tr}$ are the bottom-left, bottom-right, top-left and top-right corners of the Allen-areas relative to the interval $p_{x,y}$.

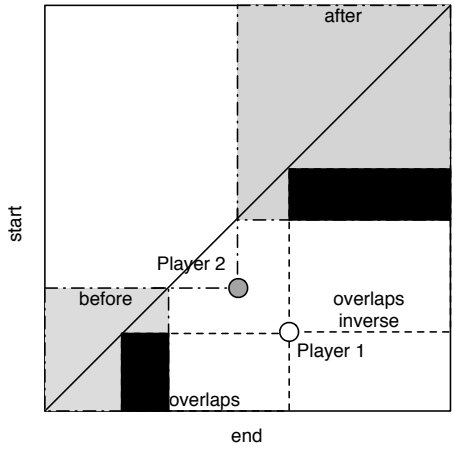
As an example, Figure 6.2a depicts two different queries. The first query (to stick with the introductory example) asks for all the teammates of a player that did not play during the tenure of another player. To query the index space, the relevant areas need to be identified. There are two temporally bound elements. Namely the tenure periods of the two players. These periods can be mapped to points in the index space. Following, the asked temporal relations can be identified. In this case it is the overlaps relations of the first player plus the before and after relations of the second player. Finally, only those regions that intersect each other (filled black areas in the figure) need to be retrieved from the hard-disk.

For a second example (Figure 6.2b), consider a query that asks for all the employees that were only employed in the duration of the merger of a company while a certain person was CEO. Again, the duration of the person holding the CEO position and the phase of the company's merger are bound. The query asks for the intersection of the two during areas of the bound intervals. Again, only the intersection (black area) is retrieved from disk.

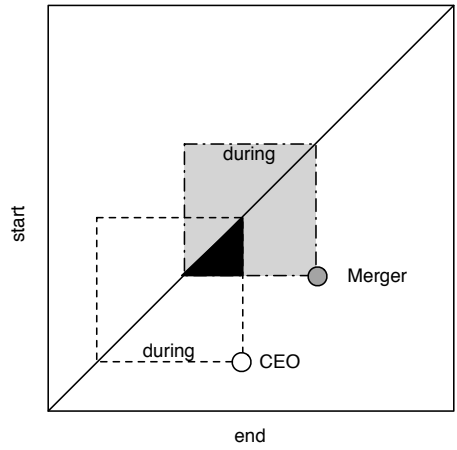
In the next section we will present our approach to map the 2-dimensional space to a single dimension using fractal curves. Followed by the presentation of our retrieval technique that weights scan vs. seek costs.

6.4 Using Space-filling Curves for the 1-dimensional Serialization

Space-filling curves such as the Hilbert-curve or the Z-curve are an elegant way to serialize a multi-dimensional space to a single dimension. Curves of this kind are categorize as FASS curves, an acronym



(a) Example 1. The intersection of the overlapping, before and after relations of two bound intervals. This is the visual form of the query: *all the teammates of a Player 1 that did not play during the tenure of Player 2.*



(b) Example 2. The intersection of the overlapping regions of two bound intervals. This is the visual form of the query: *all the employees that were only employed in the duration of the merger of a company while a certain person was CEO.*

Figure 6.2: Two query exaples. The solid black areas are the regions containing the elements that match the query.

that stands for: space-filling, self-avoiding, simple and self-similar. The fact that a curve is self-avoiding is crucial as it guarantees that during its traversal through space it will never visit the same point twice. The predicate simple denotes a small set of replication rules which produce the curve (mostly recursively). The Hilbert curve is known for its superior locality characteristics: the fact that two consecutive points in the traversal order also have a high closeness in the higher-dimensional space. In Figure 6.3 the first three resolutions of both Z- and Hilbert-curve are presented. The fact that the Z-curve leaps at certain positions in its order is the rationale behind the inferior locality preserving characteristic.

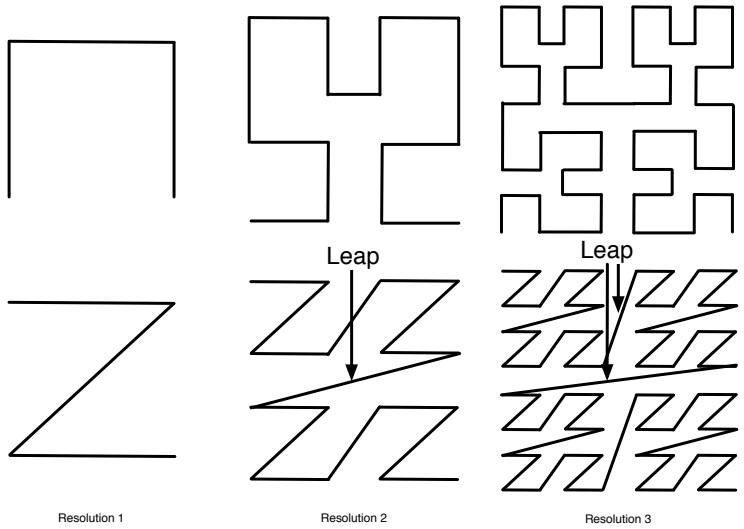


Figure 6.3: The first three resolutions of the Hilbert- and Z-curve.

Nevertheless, solely the Z-curve has come to application for multidimensional indexing (UB-Tree) mostly due to the lack of efficient

calculation algorithms for the Hilbert order and the range-retrieval. Naturally, the algorithms to encode and decode the Hilbert order are more complex because the replication pattern of the Hilbert-curve contains more mutations than the one of the Z-curve. Recently, Chen *et al.* [Chen et al., 2007] presented a method to encode and decode the Hilbert order with the time complexity of $O(r_{min})$. r_{min} is the resolution of the Hilbert curve that contains the asked point. In practice, when indexing the whole space that can be mapped to a 64-bit integer number, r_{min} is $log_2(2^{64}) = 64$. This implies that for a 64-bit index space, the calculation of the Hilbert order takes 64 iterations in the worst case. In our opinion, this is a negligible amount of computing effort in current computer systems.

In the next section, we present our deterministic range retrieval algorithm for the Hilbert curve. This will, combined with our mapping approach (Section 6.3) and Chen *et al.*'s algorithm, lay the foundation of the Hτ-index.

6.5 The Hτ-Index Range Retrieval Algorithm

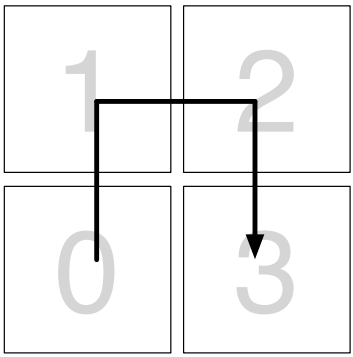
Range retrieval algorithms for the Hilbert-curve are scarce. For the sake of the retrieval of rectangular ranges from large images, Song and Roussopoulos [Song and Roussopoulos, 2002] described a technique to retrieve a rectangle from a Hilbert curve. Their method consists of the in-memory creation of a set with all the points that are located on the hull of the query rectangle. For every point, they calculate the Hilbert-order. Next, they take the previous and next number of the Hilbert order and decode it to a 2-dimensional point. If both, the next and previous points are within the query rectangle, the initial point is removed from the set. After the procedure terminates (*i.e.* every point on the rectangle's hull visited), the set contains

only those points where the Hilbert curve enters or exits the query rectangle. As a final step, the elements need to be ordered and iterated. Adapted to our index structure, the number of iterations in this set relates to the number of seek operations that need to be executed in the B+Tree. As an optimization step towards the reduction of disk accesses, the authors proposed the extension of the query rectangle to the quadrant borders. The initial query box extended to align with full quadrants of the Hilbert order. The quadrant borders are located wherever the x and y coordinate are divisible without remainder with the number 2^n where n is a number between 1 and r_{min}. As a result, there are r_{min} different optimization possibilities—a brute-force approach. Combined with the expensive set-based retrieval algorithm, the time complexity grows to $O(6 * (w + h) * r_{min}^2)$ with w and h being the width and height of the query rectangle. Next, we will present our novel deterministic range-retrieval algorithm having a built in cost-function and the superior worst-case complexity of $O((w + h) * r_{min})$.

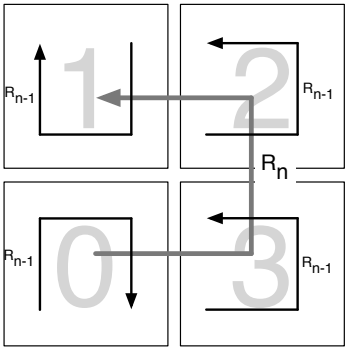
Our algorithm can be classified as a divide-and-conquer approach as it starts on the top-level (at resolution r_{min}) and then decides whether the costs for the retrieval of the top level are lower than the costs of splitting the query area and retrieving the sub-levels. Such a decision involves paying for the resulting additional seek operations but has the benefit of less retrieval of overhead elements. While the Z-curve traverses the quadrants in the same way in every resolution. Numbering the quadrants in a mathematical way (0 = bottom-left, 1 = top-left, 2 = top-right, 3 = bottom-right), the traversal order of the Z curve is 1–2–0–3, whereas the Hilbert curve changes the traversal order depending on the order of the parent resolution. To that end, Table 6.2 visually lists the Hilbert orientations in the subquadrants according to the parent orientation

and serves as a lookup table for our algorithm. As an example of the application of Table 6.2 consider the Figures 6.4a and 6.4b. In Figure 6.4b the orientation is east (E) because the middle two quadrants of the traversal are located on the left-hand or east side. The rotation is counter-clockwise (CCW). To determine the orientation and rotation of the next deeper level, we use Tabel 6.2. As an example, consider quadrant 1 of Figure 6.4b. To determine its direction and orientation we have to consider the fifth column where the orientation of the current resolution is listed (E). Next we go to the 3rd row where the values for quadrant 1 are listed. We can read the orientation south (S) and the orientation clockwise (CW). This procedure can be applied for every position of the Hilbert order to determine the traversal order of the next lower resolution.

Another feature of the Hτ-index range retrieval algorithm is the fact that it passes information about connecting quadrants to the next lower level. It may pay off to retrieve a higher-level quadrant and not breaking the incoming Hilbert order instead of digging down to the next level and losing the incoming connector (*i.e.* paying for a seek) but retrieving less elements. This decision is the domain of the built-in cost function. In the following, the calculation steps are described textually. Additionally, Algorithms 1 and 2 line out the procedure in pseudo-code.



(a) An example of a traversal direction of the Hilbert order. In this example, the traversal orientation is north (N), because the middle two quadrants are the upper quadrants of the square. The traversal direction is clockwise (CW).



(b) An example of the lookup Table 6.2. The parent orientation is east (E) and the rotation is counter-clockwise (CCW). According to the lookup table, the orientations and rotations of the quadrant 0 is N and CW, quadrant 1 is S and CCW and quadrants 2 and 3 are E and CCW.

Figure 6.4: Examples for the different orientations and rotations of the Hilbert order.

parent orientation	⊓ N(orth)	⊏ W(est)	⊔ S(outh)	⊐ E(ast)
parent rotation	CW CCW	CW CCW	CW CCW	CW CCW
quadrant 0	⊐ CCW CW	⊏ CW CCW	⊔ CW CCW	⊓ CCW CW
quadrant 1	⊓ CW CCW	⊏ CW CCW	⊐ CCW CW	⊔ CCW CW
quadrant 2	⊓ CW CCW	⊔ CCW CW	⊏ CCW CW	⊐ CW CCW
quadrant 3	⊏ CCW CW	⊓ CCW CW	⊔ CW CCW	⊐ CW CCW

Table 6.2: Lookup table for determining the traversal direction of the Hilbert order given the child quadrant and the parent orientation. CW and CCW stands for clockwise and counter-clockwise.

6.5.1 Step 1 - Setup

As a preparation step for the algorithm the value r_{min} needs to be calculated. r_{min} is the minimal dimension of the Hilbert curve that encloses the query region entirely. It can be calculated using the coordinates of the bottom-left and top-right corner point of the query region as depicted in Formula 6.1.

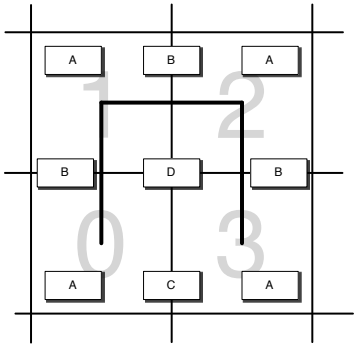
$$r_{min} = \lceil max\{ln(tr_x), ln(tr_y)\}\rceil \tag{6.1}$$

Next, it is determined whether the query rectangle intersects more than one quadrant of the resolution r_{min}. If yes, the value of r_{min} is increased by 1 in order to assure that the query rectangle lies completely within one quadrant of the initial resolution—the input requirement for the subsequent steps. Since r_{min} is the resolution of the Hilbert order that covers the area of from the source point (0,0) to a point $>$ the top-right point of the query rectangle, we can exploit the fact that the Hilbert curve only has two initial orientations and rotations. Whenever r_{min} is an even number, the Hilbert curve of that resolution rotates counter-clockwise with the traversal order 0–3–2–1. Is it an odd number it rotates clockwise and traverses 0–1–2–3. Having put this information together, the algorithms iteration can be initialized starting with Step 2.

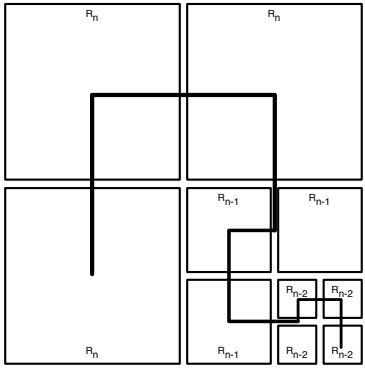
6.5.2 Step 2 - Square Type Distinction

In order to facilitate the distinction of the subsequent calculation steps we identified four different cases. Depending on how many quadrants a query rectangle intersects the algorithm follows a different strategy. Figure 6.5a depicts the different rectangle types. Type A is completely contained by a quadrant. Type B intersects two quadrants, but those quadrants are connected in the traversal direction

of the Hilbert curve. Type C is also contained in two quadrants, but those quadrants are not directly connected by the Hilbert curve. Finally, Type D intersects all four quadrants.



(a) The four different area types in the Hilbert order. Type A does not cross a quadrant border, type B crosses a quadrant border in the traversal direction. Type C crosses a quadrant border, but not in traversal direction and type D overlaps all four quadrants.



(b) The traversal order of the Hτ-Approach. At every quadrant the costs are evaluated for splitting the query region into sub-quadrants.

Figure 6.5: Area types and the traversal order of the Hilbert curve when walking down different resolutions.

The distinction between the different rectangle types requires knowledge of the orientation and traversal order of the Hilbert curve in the current resolution and quadrant. Table 6.2 lists the different possible orientations and rotations based on the information about the parent resolution. It serves as a lookup table for the algorithm in the subsequent steps.

Should the current resolution equal zero, then the current space is retrieved (important as a terminating step for the recursion, see

below). Once the rectangle type is known, the algorithm can divide the rectangle area and step into a deeper resolution.

RECTANGLE TYPE A (LINE 4 OF ALGORITHM 2)

If the square is of type A (rectangle query completely covered by a single quadrant) the algorithm decides whether to retrieve the whole query space (four quadrants) or focusing on the single quadrant and further divide the query space. On the first sight it may look obvious that it is always beneficial to further divide the query rectangle. In fact, it depends whether the quadrant would be omitted where the Hilbert order enters the query region from the higher level. The dropping of such a connecting quadrant comes at the cost of paying for a additional seek operation. The cost function weights between paying these costs and the alternative case (not having an additional seek but retrieving overhead elements that later need to be filtered). The costs c_{scan} of retrieving the current space is dependent on whether there is a connector from the upper level ((Line 6 of Algorithm 2)). If yes, $c = 0 * c_{seek}$ else $c = c_{seek}$ where c_{seek} is the cost of one seek operation. The scan costs are $c = c_{scan} * A_{space}$ being the area of the current query space. On the other hand, the costs for stepping to the next resolution is the risk of losing a connector. For the drop of either an in-connector or an out-connector c_{seek} is added to the costs. The benefit is the reduced overhead-retrieval since the scan costs are $c = A_{quadrant} = 1/4 * A_{space}$. Depending on which costs are smaller, the algorithm can either retrieve the query space (and terminate that branch of the recursion) or go deeper to the next lower resolution level with the affected quadrant as the input query space.

RECTANGLE TYPE B (LINE 15 OF ALGORITHM 2)

A bit more complex than type A are type B rectangles. Those rectangles span over two quadrants with the quadrants being connected in the Hilbert traversal order. Again, the decision has to be made whether to retrieve the current query space or stepping to a lower resolution and only get the two affected quadrants. The cost calculation for the first case is equals to the calculation for type A rectangles. The costs for the second case are also dependent on the presence of a possible in- and/or out-connector. For the presence of each of them, the costs of an additional seek arises if the affected quadrants do not intersect with the connectors. Again, the advantage of going one level deeper is the reduced scan costs being in this case $c = A_{quadrant} = 1/2 * A_{space}$ (Line 18 of Algorithm 2). Should it be beneficial go to the next resolution, then the query space needs to be split up at the quadrant border into two different regions. The algorithm continues at Step 2 with the first quadrant (in traversal order) and its respective piece of the query region as parameters. The information about a (possible) in-connector, as well as the fact that there is an out-connector has to be passed to the next step. Once this step terminates, it can be started again with the second quadrant and a possible out-connector and in-connector.

Rectangle Type C (Line 28 of Algorithm 2)

The decision procedure for Type C rectangles is less complicated. It does not depend whether there are in- and out-connectors from the higher level. Those quadrants are retrieved in any case. The cost function only has to distinguish whether the costs of an additional seek (when only retrieving the two affected quadrants) is lower than the costs for the additional scans when retrieving the whole query space $c = A_{quadrant} = 1/2 * A_{space}$ (Line 31 of Algorithm 2). If yes, the query space is retrieved as a whole, if no, the query rectangle is

split according to the affected quadrant borders and the algorithm continues at Step 2 with the new parameters. Possibly present in- and out-connectors are passed on to the lower level accordingly.

RECTANGLE TYPE D (LINE 41 OF ALGORITHM 2)

Type D rectangles cover all four quadrants of the query space. The decision on whether to go one level deeper is only dependent on the amount of overhead that would be retrieved on the current level. If the costs of this overhead are less than the costs for an additional seek, the algorithm retrieves the current query space. Otherwise, the query rectangle is split into four parts (according to the quadrant borders) and the algorithm continues at Step 2 with the quadrants and their respective part of the query rectangle. Possible in- and out-connectors are passed to the next level accordingly.

6.5.3 Step 3 - Retrieval

Once the above iterations terminate, the elements can be retrieved from the underlying B+Tree structure. In order to optimize the lengths of the seek operations, the partial areas need to be connected again (it may be the case that a higher level resolution area is connected to a lower-level one). To that end, the entry and exit values (*i.e.* the translation of a point to its respective number in the Hilbert order) for every area need to be listed. Next, it is checked whether there is either (1) another entry value minus 1 that is equals to the exit value or (2) another exit value plus 1 equals to the entry value. If they match, they can be combined to a single entry / exit value pair. This results in the optimal set of start and end values for the retrieval from the B+Tree according to the given costs for scan and seek operations.

In the next section we will compare our algorithm to other approaches both qualitatively and quantitatively.

Algorithm 1 $setup(bl_{x,y}, tr_{x,y})$

1: $res \leftarrow \lceil max\{ln(tr_x), ln(tr_y)\}\rceil$ {determine r_{min} and initialize resolution (res)}
2: **if** $squaretype(bl_{x,y}, tr_{x,y}, res) \neq A$ **then**
3: $res \leftarrow res + 1$
4: **end if**
5: **if** $modulo(res, 2) = 0$ **then**
6: $rotation \leftarrow CCW$
7: $orientation \leftarrow E$
8: **else**
9: $rotation \leftarrow CW$
10: $orientation \leftarrow N$
11: **end if**
12: $decide(space, query, rotation, orientation, false, false)$

Algorithm 2 $decide(query, space, p\_orientation, p\_rotation, in\_connector, out\_connector)$

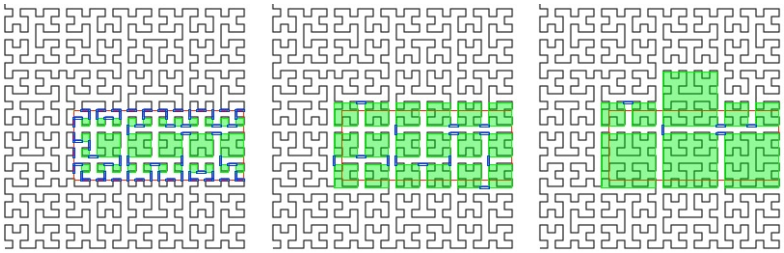
1: $q[] \leftarrow determine\_quadrant()$ {Those quadrants of $space$ that are intersected by the $query$}
2: $c\_orientation \leftarrow lookup\_orientation(p\_orientation, p\_rotation)$ {Information from Table 2}
3: $c\_rotation \leftarrow lookup\_rotation(p\_orientation, p\_rotation)$
4: **if** $rect\_type == A$ **then**
5: $c\_current \leftarrow space.area + c_{scan}$
6: $c\_current \xleftarrow{+} in\_connector == q?0 : c_{seek}$ {addition assignment}
7: $c\_sub \leftarrow \frac{1}{4} * space.area * c_{scan}$
8: $c\_sub \xleftarrow{+} in\_connector == q?0 : c_{seek}$
9: $c\_sub \xleftarrow{+} out\_connector == q?0 : c_{seek}$
10: **if** $c\_sub < c\_current$ **then**
11: $decide(intersection(q, query), q, c\_orientation, \_rotation, new\_out\_connector)$ {Dive into recursion}
12: **else**
13: $retrieve(q)$ {Retrieve from disk and end this branch of the recursion}
14: **end if**
15: **else if** $rect\_type = B$ **then**
16: $c\_current \leftarrow space.area + c_{scan}$
17: $c\_current \xleftarrow{+} in\_connector == q?0 : c_{seek}$
18: $c\_sub \leftarrow \frac{1}{2} * space.area * c_{scan}$
19: $c\_sub \xleftarrow{+} in\_connector == q[0]?0 : c_{seek}$
20: $c\_sub \xleftarrow{+} out\_connector == q[1]?0 : c_{seek}$
21: **if** $c\_sub < c\_current$ **then**
22: $decide(intersection(q[0], query), q[0], c\_orientation, c\_rotation, new\_out\_connector)$ {Dive into recursion of the first quadrant}
23: $decide(intersection(q[1], query), q[0], c\_orientation, c\_rotation, new\_out\_connector)$ {Dive into recursion of the second quadrant}
24: **else**
25: $retrieve(q[0])$
26: $retrieve(q[1])$
27: **end if**

Algorithm 2 decide *(continued)*

28: **else if** $rect\_type = C$ **then**
29: $c\_current \leftarrow space.area + c_{scan}$
30: $c\_current \xleftarrow{+} in\_connector == q?0 : c_{seek}$
31: $c\_sub \leftarrow \frac{1}{2} * space.area * c_{scan}$
32: $c\_sub \xleftarrow{+} in\_connector == q[0]?0 : c_{seek}$
33: $c\_sub \xleftarrow{+} out\_connector == q[1]?0 : c_{seek}$
34: **if** $c\_sub < c\_current$ **then**
35: $decide(intersection(q[0], query), q[0], c\_orientation, c\_rotation, new\_out\_connector)$ {Dive into recursion of the first quadrant}
36: $decide(intersection(q[1], query), q[0], c\_orientation, c\_rotation, new\_out\_connector)$ {Dive into recursion of the second quadrant}
37: **else**
38: $retrieve(q[0])$
39: $retrieve(q[1])$
40: **end if**
41: **else if** $rect\_type = D$ **then**
42: **if** $(space.area - query.area) * c_{scan} < c_{seek}$ **then**
43: $decide(intersection(space, query), q[0], c\_orientation, c\_rotation, new\_in\_connector, new\_out\_connector)$
44: $decide(intersection(space, query), q[1], c\_orientation, c\_rotation, new\_in\_connector, new\_out\_connector)$
45: $decide(intersection(space, query), q[2], c\_orientation, c\_rotation, new\_in\_connector, new\_out\_connector)$
46: $decide(intersection(space, query), q[3], c\_orientation, c\_rotation, new\_in\_connector, new\_out\_connector)$
47: **else**
48: $retrieve(space)$
49: **end if**
50: **end if**

6.6 Qualitative Analysis of the Deterministic Hilbert Rectangle Retrieval Algorithm

As described in the text and pseudo-code in the previous Section 6.5, the deterministic algorithm outperforms the currently available alternatives from a complexity point of view. Additionally, it generates results that are superior to the ones of the competing algorithms in terms of retrieval costs. In order to exemplify this superiority we illustrated one examples for the retrieval of rectangles from the index space. For the example query we visually outline the area that each algorithm would choose to retrieve from the B+Tree. Also, we report the number of seek and scan operations that would be needed for the retrieval of the highlighted areas.



(a) The exact retrieval of the query rectangle: a seek operation results every time the Hilbert curve enters the query rectangle from outside.

(b) The optimized approach: by expanding the query rectangle the number of seeks can be reduced at the costs of additional scans.

(c) The deterministic approach of the Hτ-index: the algorithm is aware of the traversal direction of the Hilbert curve and can weight seeks vs. scans.

Figure 6.6: Three different ways to retrieve a rectangle from the Hilbert curve. Red rectangle: the queried region, green rectangles: the effectively retrieved areas, blue: the connectors of different resolutions of the Hilbert order.

As we already pointed out: Our algorithm's major advantage is its reduced computational complexity through its awareness of the Hilbert curve's traversal order. Nevertheless, this results in an advantage in terms of the reduction of retrieval costs. The algorithm decides in every quadrant whether it should further split down the retrieval or get the current quadrant. This can optimize the retrieval locally while the optimization proposed by [Song and Roussopoulos, 2002] operates at a global level. Figure 6.6 shows an example query for the retrieval of a randomly-chosen 8-by-23 rectangle using a cost function of 1:100 (a seek operation exceeds the costs of a scan operation by a factor of 100). In Figure 6.6a, which depicts the retrieval of the exact rectangle, the number of seeks are 21 with 184 scans totaling in the retrieval costs of 2284. Figure 6.6b vastly reduces the seeks to 4 by increasing the scans to 230 with the total costs of 630. Figure 6.6c (the hτ-index) further optimizes the retrieval by reducing another seek operation (3) with an increase of scans to 258 with resulting total costs of 558. Every further optimization would result in a decrease of seek operations which would not pay off the additional scan operations (for this specific case with a cost function of 1:100). This example illustrated a single example of a query. In the next section we will present a quantitative analysis of the different approaches as well as a comparison to other temporal index approaches.

6.7 Evaluation of different retrieval strategies and index structures

As mentioned in Section 6.3, by using the end-start mapping, all possible query tasks can be achieved by the retrieval of rectangles

from the 2d index space. In this section, we will evaluate the retrieval costs of different serialization strategies (*i.e.* Z-Order, Hilbert, Map21) with respect to the scan and seek operations. In order to gain a complete picture we ran a brute-force evaluation of different query possibilities.

6.7.1 Experimental Setup

As a characteristic of space-filling curves, and especially the Hilbert curve, the number of seeks is highly dependent on both, the size and the position of the query rectangle in the index space. To avoid the risk that we only compare a (possibly biased) subset of query rectangles we apply the following procedure: First, we choose a square with the side length of l_s in which we conduct the element retrieval. In this square we start with the retrieval of the query rectangle with the length and width of 1. This rectangle is moved to every possible position in the square. At each position we calculate the number of seeks and scans and the resulting retrieval costs. In a next step, the query rectangle is extended in the width by +1. When all widths have been retrieved the height is extended by 1. This procedure results in the retrieval of every possible rectangle in the query space. The number of retrieval operations grows exponentially with l_s of the investigated square according to the formula $(1/2 * l_s * (l_s - 1))^2$. For our experiments we chose a square with the side length 100, resulting in a total of 2.35 billion different queries.

To condense this huge amount of different queries to a comprehensive figure, we calculated the average of number of seeks, number of scans and retrieval costs for each rectangle size. To capture the variation given by the different positions we report the standard deviation for every rectangle size. For the visualizations we chose a 3-dimensional graph which plots the query square's width and

height on the x and y axis and the average $c_r^{width,height}$ on the z axis. In addition, $\pm var_{c_{seek}^{width,height}}$ (the variance) is plotted in light color above / below the average values.

6.7.2 Evaluation of the different retrieval strategies

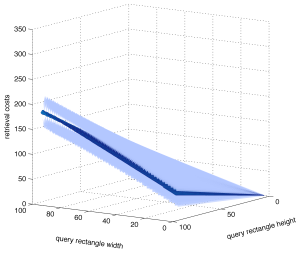
In a first experiment we retrieved the exact query rectangle from the different index structures. Therefore, no overhead retrieval is generated. For the retrieval from the Hilbert curve, we use the algorithm proposed by [Song and Roussopoulos, 2002]. The algorithm builds and sorts a list that contains the Hilbert number of all the points on the outer hull of the query rectangle. For each of these points, the algorithm compares whether the next and previous point in the Hilbert order resides inside or outside the rectangle. In this way, the points where the Hilbert order enters and exits the query rectangles can be determined. Between entry and exit point, the elements can be retrieved from the underlying B+Tree with a scan using 1 seek operation. We applied the technique to retrieve the elements from the Z order as presented by Tropf and Herzog [Tropf and Herzog, 1981]. As a baseline we compare the different space-filling approaches to the MAP21-index [Nascimento and Dunham, 1999]. Note that different optimizations for the MAP21 approach have been proposed including parallelization of disk access. We believe that such optimizations could also be applied to our Hτ-Index. Since we want to compare the basic idea of the index structures we did not consider those optimizations. For the following cost calculations we used a cost ratio of 1:100 between scan and seek costs. The Figures 6.7 (b), (a) and (c) visualize the average retrieval costs. As expected, the Hilbert curve exhibits its superior locality preserving behavior as it outperforms the Z curve in all respects. Also, the Map21 approach

generates retrieval costs that are in the region of the ones of the unoptimized Hilbert order.

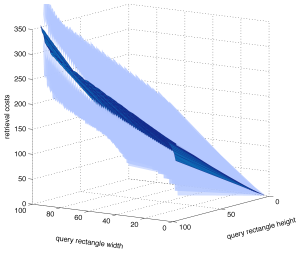
6.7.3 Optimizing Retrieval Cost by Expanding the Query Rectangle

As an optimization step towards the reduction of the number of seeks, one can make use of the properties that are characteristic to the Hilbert and the Z curve: Their patterns reproduce themselves in in a quadrant system where the pattern is replicated with modifications in every quadrant of the next resolution. The expensive retrieval (in terms of number of seeks) happens whenever the query rectangle crosses the curve in quadrants of a low resolution. An optimization (described by [Chen et al., 2007] for image processing) towards the reduction of seek operations is the expansion of the query rectangle until it reaches the quadrant borders of a lower resolution. The calculation for the The extension of the query space can be done until it covers a whole quadrant of a lower resolution. Such a quadrant can then be retrieved with the minimum of one seek operation. Since an overhead of elements are retrieved from the index, the additional elements cause the direct costs of reading them from the index structure and the indirect costs of filtering in a processing system. We combine both, the retrieval plus the filtering in the cost variable c_{scan}. For every expansion to the borders of a resolution, we can measure n_{seek} and n_{scan} and then compute c_r by multiplying them with the cost ratio of scans and seeks. A final iteration over the set of resolution–cost pairs can determine the resolution with the lowest costs an retrieve the rectangle that is expanded to this resolution.

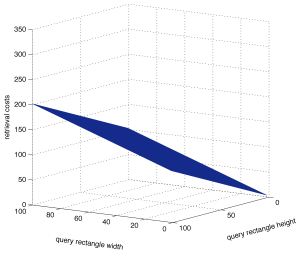
Figure 6.7d shows the retrieval costs generated by the Hilbert curve using this optimization. Figure 6.7e shows the costs of the



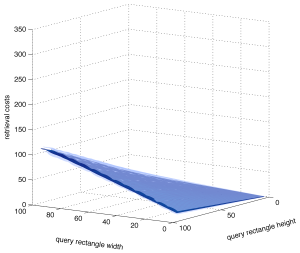
(a) The Hilbert retrieval costs using the algorithm proposed by [Song and Roussopoulos, 2002]



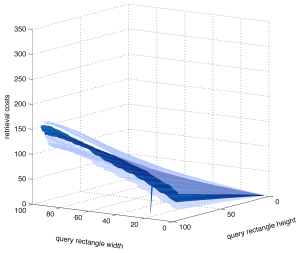
(b) The Z-Curve retrieval costs



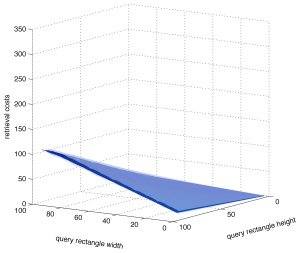
(c) The retrieval costs for the Map21 index. Note: Map21 creates for every interval two entries in the index resulting in a space consumption of $2 * n$.



(d) The optimized retrieval from the Hilbert curve using the algorithm by [Chen et al., 2007].



(e) A similar optimization to the one presented by [Chen et al., 2007] (for the Hilbert curve) applied to the Z curve.



(f) The deterministic algorithm's cost and standard deviation

Figure 6.7: The costs of the different retrieval strategies.

Z curve using the aforementioned optimization. We can observe a reduction of the retrieval costs of about 50 percent in both the Z and Hilbert cases. Both the Z and Hilbert curve now outperform the MAP21 approach. In the next section we will evaluate our further optimizations applied to the Hilbert curve, the Hτ-Index.

6.7.4 The Retrieval Costs when Using the Hτ Algorithm

The downside of the above optimizations are their non-deterministic nature. In other words, every possible candidate-optimization has to be calculated. After the calculation, the cost function can be applied to see which variant results in the least costs. As mentioned, this increases the computational complexity for the Hilbert curve from $O(6*(w+h)*r_{min})$ to $O(6*(w+h)*r_{min}^2)$. Also, the optimization does not result in the optimal rectangle in terms of minimal costs. In our last evaluation we are going to show the cost for the retrieval from the Hτ-Index. Note that the improvement of the retrieval costs (3.35%) seems low, but the major improvement is the reduced computational complexity of $O(max\{w,h\}*r_{min})$.

Also, the variance is reduced by 20.9% compared to the optimized version of the above section. The implications of this lower variability are a more reliable behavior of the index structure in terms of query answering time of the (RDF-)storage system. To sum it up, the Hτ-Index brings a huge reduction of retrieval costs compared to the Z-curve (by about 25-50%, depending on the optimization). Compared to the Map21 approach, the Hτ-Index retrieval cost are lower by almost 50% which in turn does not have a variance in retrieval costs.

6.7.5 Retrieval Costs with Different Cost Ratios

So far, we used only a cost ratio between scans and seeks of 1 to 100 which is typical. In order to evaluate different cost ratios we ran the retrieval algorithms with different cost functions. Since the general shape of the above figures remains the same, we will only report the relative improvement of the algorithms compared to the Map21 approach. We listed the values in a comprehensive table. In Table 6.3 the retrieval costs of Map21are listed in the second column as a baseline. In the remaining columns the relative improvement (*i.e.* cost reduction) is listed for every algorithm (exact Hilbert, optimized Hilbert, $H\tau$, Z, optimized Z). The higher the number, the higher the cost reduction compared to Map21.

c_{scan}:c_{seek}	c_r	*relative improvement-factor compared to column 2 (c_r of Map21)*				
Ratio	**Map21**	**Hilbert**	**Hilbert Optimized**	**Hτ**	**Z**	**Z Optimized**
1:1	10102.00	8.28	7.46	8.28	**8.41**	7.96
1:10	1012.00	4.30	4.26	**4.39**	3.47	4.16
1:20	507.00	3.30	**3.88**	3.83	2.37	3.40
1:30	338.67	2.76	3.51	**3.56**	1.92	2.99
1:40	254.50	2.42	**3.33**	**3.33**	1.60	2.70
1:50	204.00	2.20	3.18	**3.24**	1.34	2.48
1:60	170.33	1.96	3.07	**3.16**	1.15	2.30
1:70	146.29	1.79	2.98	**3.07**	1.03	2.15
1:80	128.25	1.66	2.90	**3.00**	0.94	2.02
1:90	114.22	1.56	2.85	**2.95**	0.86	1.91
1:100	103.00	1.43	2.79	**2.89**	0.78	1.82

Table 6.3: The resulting retrieval costs of the different algorithms with changing ratio of scan vs. seek costs (best algorithm in bold-face).

As a reading aid for Table 6.3 consider the row with ratio 1:50. There, the cost for the retrieval from the Map21 index is 204. The retrieval of the exact Rectangle from the Hilbert curve has 2.2 times less cost, the retrieval using the Hτ-Index leads to a cost reduction by factor 3.24. It is interesting to observe that with an increasing cost ratio the improvement decreases. This is due to the fact that the Map21 index optimizes the disk access to two seek operations with a (more or less) huge amount / length of scan operations. With increasing costs of the seek compared to a scan, there is a break-even point where the seek-costs compensate the huge scan-overhead. This ratio is above any realistic number for existing hard-disk drives, and therefore we believe that it has no implications for real-world applications.

6.8 Limitations and Future Work

Our approach has certain disadvantages which can reduce its applicability under certain circumstances. A problem that could arise is the computational complexity. We could successfully reduce the computing effort needed to calculate the index numbers of the Hilbert order that need to be retrieved from the B+Tree. Recently, Chenyang *et al.* [Chenyang et al., 2008] presented a new algorithm to calculate the Hilbert order which they show has an equal performance as the calculation of the Z order. This overthrows the last advantage of the Z curve over the Hilbert curve.
Furthermore, our current algorithm does not optimize the disk accesses in terms of index population. So far, we assumed the index to be fully populated which is a non-practical model-assumption. In a practical system, the algorithm should be equipped with a side-structure in the form of a load matrix that stores information about

whether quadrants (of a certain resolution) contain index elements. Before the algorithm accesses the hard-drive, it should consult the load matrix whether there is any chance to encounter index elements in the (sub-) areas of the query rectangle.

Derived from the limitations, future work will have to investigate the implementation of the index structure in a triple-store to assess its performance under real-world conditions. We believe that the hτ-index can vastly improve query answering times for temporal graph-patterns and provide new operators that make use of the Allen interval relations to facilitate the composition of more intuitive queries.

6.9 Conclusions

In this paper we presented the Hτ-Index. An index structure for temporal data based on the Hilbert space-filling curve. By mapping an interval's start and end time to the 2-dimensional space, we are able to assign each Allen-relation a specific area in the parameter space. When querying for temporal (possibly intersecting) relations, the query can be expressed as a rectangle which then can be retrieved from the index. The index is a standard B+Tree which holds the 2-dimensional space which is serialized using the Hilbert order. In a second part of our work we presented a novel approach of retrieving these query rectangles from the Hilbert order which optimizes the disk accesses of the underlying B+Tree. This algorithm outperforms the the so-far known retrieval strategies in terms of retrieval costs. Additionally, it has a superior computational complexity compared to other algorithms such as the one presented by Chen *et al.* [Chen et al., 2007]. Since our approach is similar to the UB-Tree which uses the Z curve for serialization, we also compared to this in-

dex structure. We could show that the Hτ-Index can provide up to 50% less retrieval costs which directly translate to query answering time.

Since graph-based data structures such as RDF do not provide a well-defined data schema it is not trivial to provide fast query answering times. We believe that our index structure can provide a building block towards the goal of a time-aware Semantic Web which does not forget information upon updating.

7

Towards Practical Temporal Reasoning: Definition and Entailment

Jonas Tappolet and Abraham Bernstein

Reviewed at the European Semantic Web Conference, ready for submission

ABSTRACT In recent research, temporal extensions to Semantic Web languages such as OWL and RDF have been proposed. As a consequence, reasoning in knowledge bases with temporal annotation has been investigated. We show that there are two different semantics of time, *Union* semantics and *Intersection* semantics. We define an extended set of RDFS entailment rules in order to apply union

semantics to RDFS. Additionally, we are presenting a new species of concepts that can be entailed from temporal information, the dynamic temporal concepts (DTC). Based on a timeline, new knowledge can be entailed depending on restrictions given by a user. We introduce 4 basic DTC-types, the duration, frequency, coverage and density. These patterns can be used to define concepts such as "Two persons are considered to be in contact when they send each other an email message at least once a week". By mapping the detection of these patterns to Timed Automata, we can prove decidability and satisfiability. We present a prototype implementation of the DTC-reasoner and demonstrate its performance.

7.1 Introduction

Time is an important factor in data representations. Oftentimes it is used to annotate data with the time of its insertion (processing time), e.g. the exact time when a newspaper issue appeared. Additionally, when introducing validity intervals, relations can be temporally quantified, for instance that a person subscribed to a newspaper from 2005 until 2010. This annotation of validities has been applied to the Semantic Web as a non-standard extension of RDF and OWL. These formalisms annotate every triple with an interval of validity. Since time and validity is represented in a unified way, automated reasoners can entail new facts from the temporal base facts. The calculus presented by Allen [Allen, 1983] enables the entailment of the relations between time intervals. Interval relations such as before, after, and overlaps (13 in total) can extract the semantic information that is buried in the numerical representation of the validity intervals. Once this information is back in the knowledge base as symbolic values, it can be further processed by a standard

reasoning system and be treated same as any other triple. Recently Tao et al. [Tao et al., 2010] have shown an application of this kind of temporal reasoning applied to the medical domain for electronic patient records. By entailing, e.g., that a complication occurred before a vaccination a reasoner can exclude the possibility that the complication is a side effect of the vaccination.

Temporal information can be further exploited when combined with the entailments that can be made in logic-based representations such as RDFS and OWL. Consider the following example: We know that Jack was born in 1980 and is still alive today. We also know that Jack is married to Jane since the year 2000. Based on background knowledge that only adults can be married to each other and adults are persons over 18, which is defined in an ontology, we can derive additional temporal information. In the above example this would be that Jane must be born in, or before 1982. But what if a class inherits temporal properties from two superclasses? How should those temporal constraints be combined?
As a first contribution of this paper *we present two different temporal semantics*—one based on an *Intersection* as previously explored [Motik, 2010] and another, novel one based on the *Union* of temporal constraints—that can be applied when combining temporal and logical reasoning and present an extension of the RDFS reasoning rules in order to entail temporal information.

A different type of temporal entailments are patterns that occur over time rather than temporal constraints. This second type of temporal pattern is used to characterize dynamic temporal phenomena such as the reoccurring sequence of seasons or the growing of a human from baby, to child, young adult, etc. We propose to reason (and or match) such patterns using a decidable subset of timed automata (event clock automata). Once such a pattern is matched, it can be

named similar to OWL equivalence classes and properties. Consequently we can, e.g., define a frequent concept `FrequentFlyer` as a human who has at least four `fliesTo` property annotated with temporal constraints within a year.
The second contribution of this paper is the *specification of* this kind of *dynamic temporal reasoning, its mapping to timed automata to prove decidability and satisfyability*, and the *definition of four basic building blocks for dynamic temporal patterns* that can be used to define a myriad of different temporal entailments. Once these named patterns are first-class citizens in the knowledge base and can participate in further entailments. Timed automata have been widely used to analyze e.g. real-time systems. [Raimondi et al., 2008] for instance used timed automata to monitor service level agreements of web services.

The remainder of this paper is structured as follows. Next, we will succinctly review the relevant literature in the field of temporal reasoning and representation. Second, we introduce the two different definitions of temporal semantics, one which we consider to be more suitable for OWL, whereas the other shares the spirit of RDFS. In Section 7.4 we will present an extended set of RDFS entailment rules for temporal reasoning along these semantics. The main part of the paper deals with the definition of the temporal patterns and their detection using timed automata. We will present a prototype implementation and evaluate it against a realistic dataset with 742,96 million triples. We conclude with limitations of our approach and future work.

7.2 Related Work

In this section we will put our work in relation to existing approaches and point out differences and similarities.

Temporal Extensions There exist various temporal extension to Semantic Web technologies on different levels of abstraction. Gutierrez et al. [Gutierrez et al., 2007] proposed a temporal extension to RDF, which laid the foundation to subsequent publications. In addition, various papers investigate the incorporation of time to the Semantic Web on an ontological level. Welty et al. [Welty and Fikes, 2006] presented an ontology for fluents in OWL. Their method allows temporal annotations on a symbolic level. However, in their model the numeric values of a temporal annotation could theoretically contradict the symbolic representation. Other temporal extensions on a ontological level, such as [Milea et al., 2008, Hyvönen, 2007], use a more application-centric model of time and validity, which has a limited generalizability.

Static Temporal Reasoning Recently, Boris Motik [Motik, 2010] presented a temporal extension to OWL. The focus of his work was a logical extension of OWL. Our alternative definition, which we introduce in the next section of temporal semantics has some characteristics of OWL. Through the definition of equivalence classes a reasoner can entail new knowledge based on the restrictions defined by the equivalence class. To limit complexity we do not allow disjoint classes, which makes our approach more akin to RDFS reasoning.

Dynamic Temporal Concepts The notion of dynamic temporal concepts has many similarities with stream reasoning, where data patterns are detected in a stream of triples arriving over time. Barbieri et al. [Barbieri et al., 2010], e.g., presented c-SPARQL, a SPARQL extension for the execution of queries over a data stream. From a temporal perspective they use a processing-time model with a timestamp annotation of the triples. Hence, it can be seen as a sub-

set of temporal reasoning, which has similar properties but uses interval-annotated triples. In order to prevent an excessive memory consumption, stream reasoners limit their processing to a time window w. Only triples that arrive within this time window (i.e., their arrival time $t \in [now - t, now]$) are considered for matching the graph pattern defined by the query that is registered using an extended SPARQL syntax.

Other than stream querying we are not aware of other approaches that entail new knowledge from temporal occurrences of triple-based information.

7.3 Union and Intersection Semantics

Most temporal extensions to OWL make the implicit assumption that a temporal annotation renders the annotated triples valid during the given interval, and, consequently, invalid during the inverse of the interval. This inverse argument—that a fact is not valid in the inverse period of its annotation—leads to a semantics of time that is different to an open world interpretation of temporal intervals. We call these two semantics the *Union* and *Intersection* semantics of time.

In order to exemplarily elaborate on the difference of these two semantics consider the following example (see also Figure 7.1): We define a small TBox with the topmost class Car with three subclass relationships to Sportscar, Porsche and 911. Additionally, there is one instance of the class 911 called Johns_911 and another instance of the type Sportscar named Marys_Bugatti_Type35. Every entity is annotated with an interval of validity. The interpretation would be, e.g., that Mary's Bugatti was built in 1925 and still exists today. As you can see, the instance Marys_Bugatti_Type35 has unclear semantics. Although it is declared to be a Sportscar, the instance

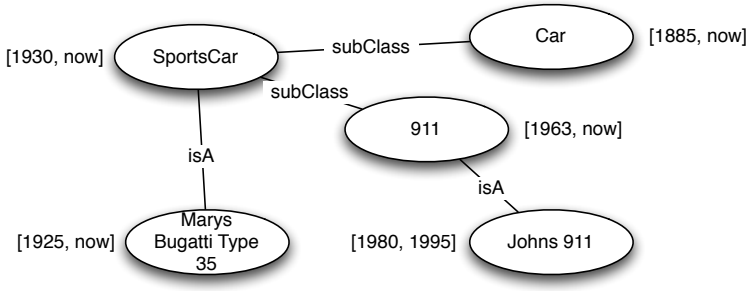


Figure 7.1: An example with two different temporal interpretations: A reasoner can either entail that Marys Bugatti was not a SportsCar between 1925 and 1930 or, it could infer that the type SportsCar has already existed in 1925. This is an example for the intersection and union semantics of time.

existed before the class that it belongs to came into being. Using *Intersection semantics* we can infer that the instance with the name Marys_Bugatti_Type35 exists since 1925, but only became a Sportscar in 1930. This is the typical valid-time interpretation of many temporal RDF / OWL approaches. It is a top-down approach that assumes eternal validity on the topmost level and further constraints the interval of validity of subsequent levels. This bears the possibility of introducing temporal inconsistencies when applying logical reasoning. To illustrate such an inconsistency consider the second example:

$$[ever,\ now]\ livesIn\ domain\ Human \tag{7.1}$$

$$[1980,\ now]\ Jane\ type\ Human \tag{7.2}$$

$$[1975,\ now]\ Jane\ livesIn\ Zurich \tag{7.3}$$

Based on triple (1) and (3) a reasoner would entail that Jane is a human from 1975 until now. Triple (2) states contradictory temporal information, namely that Jane was only a human starting 1980. When applying intersection semantics this example KB becomes inconsistent as we assume that a statement evaluates to false outside its interval of validity.

A *Union semantics*, in contrast, would entail that the union of both intervals is applicable and that Jane must have already been of type human also between 1975 and 1980.

Obviously, it is a philosophical argument, which interpretation of temporal semantics is correct. Intersection Semantics, are more expressive since they allow negation at the cost of introducing inconsistencies. In other words, intersection are a type of *iff*-semantics where union are a kind of *if*-semantics.

Formal Definition Based on the definitions and symbols for RDF graphs with temproal semantics by Gutierrez et al. [Gutierrez et al., 2007] a temporal triple is a common RDF triple annotated with a temporal validity in the form: $\langle s\ p\ o\rangle\,[i_1, i_2]$ with $i_{1,2} \in \mathbb{N}$ and $i_1 \leq i_2$, where the validity of a triple is denoted by the interval $[i_1, i_2]$. Hence, $\langle s\ p\ o\rangle\,[i] \equiv \langle s\ p\ o\rangle\,[i, i]$. To formally distinguish between the union semantics and the intersection semantics we can write:

$$\langle s\ p\ o\rangle\,[i_1, i_2] \underset{\rightarrow}{\cap} \langle s\ p\ o\rangle\,[i_1, i_2] \wedge \neg\,\langle s\ p\ o\rangle\,[-\infty, i_1] \wedge \neg\,\langle s\ p\ o\rangle\,[i_2, +\infty] \quad (7.4)$$

$$\langle s\ p\ o\rangle\,[i_1, i_2] \underset{\rightarrow}{\cup} \langle s\ p\ o\rangle\,[i_1, i_2] \wedge\ (\neg\langle s\ p\ o\rangle\,[-\infty, i_1]) \vee\ \langle s\ p\ o\rangle\,[-\infty, i_1])\ (7.5)$$

$$\wedge\ (\langle s\ p\ o\rangle\,[i_2, +\infty] \vee\ \neg\,\langle s\ p\ o\rangle\,[i_2, +\infty]) \quad (7.6)$$

Formula (4) defines the *Intersection* semantics which negate the validity of a triple outside the annotated interval whereas the *Union*

(5,6) semantics leave the evaluation of a truth value outside the annotated interval open. The strict negation of the intersection semantics are the reason for possible inconsistencies. A graph is a set of triples $\langle s\ p\ o\rangle \in G$ and, consequently a temporal graph $\langle s\ p\ o\rangle\ [i_1, i_2] \in G_\tau$ is a set of temporal triples. If function $untime(x)$ removes the temporal information the we can define the following relations. When $S_\tau \subseteq G_\tau$ is the set that $RDFS_\tau$-entails G_τ and $S \subseteq G$ $RDFS$-entails G, we can define the notion of static temporal reasoning and dynamic temporal reasoning as: $S_\tau \models_{\tau_{static}} G_\tau$ where $\models_{RDFS} \subset \models_{\tau_{static}}$ and $S \models_{RDFS} G$ then $untime(S_\tau) = S$. This describes the pure operation of static temporal reasoning on the validity annotation of a graph (while not entailing new triples). Consequently, dynamic temporal reasoning can be defined as $D_\tau \models_{\tau_{dynamic}} G_\tau$ with $S \subseteq D_\tau$ and $untime(D_\tau) \nsubseteq S$.

7.4 Static Temporal Reasoning

Given the above definition of temporal semantics this section will provide a temporally extended set of entailment rules for RDFS. These extended rules entail temporal validities using union semantics. Static temporal entailments do not add additional triples to the knowledge base (as opposed to dynamic temporal reasoning). Therefore, the added triples of the below entailment rules are pure RDFS entailments. The temporal rule "payload" does only manipulate the temporal validity of a triple. As intended by the Union semantics, the temporal validity of an updated triple always includes its validity before the update. In Table **??** we used the revised set of RDFS entailment rules as presented by H.J. ter Horst [ter Horst, 2005] which we extended with temporal union semantics rules.

	If G contains	Where	Then add to/update G
lg	$\tau\ v\ p\ l$	$l \in L$	$\tau\ v\ p\ b_l$
gl	$\tau\ v\ p\ b_l$	$l \in L$	$\tau\ v\ p\ l$
rdf1	$\tau\ v\ p\ w$		$\tau\ p$ `type Property`
rdf2-D	$\tau\ v\ p\ l$	$l = (s, a) \in L_D^+$	$\tau\ b_l$ `type` a
rdfs1	$\tau\ v\ p\ l$	$l \in L_p$	$\tau\ b_l$ `type Literal`
rdfs2	$\tau_1\ p$ `domain` u		$(\tau_1 \cup \tau_2) p$ `domain` u
	$\tau_2\ v\ p\ w$		$\tau_2\ v$ `type` u
rdfs3	$\tau_1\ p$ `range` u		$(\tau_1 \cup \tau_2)\ p$ `range` u
	$\tau_2\ v\ p\ w$	$w \in U \cup B$	$\tau_1\ w$ `type` u
rdfs4at	$\tau\ v\ p\ w$		$\tau\ v$ `type Resource`
rdfs4bt	$\tau\ v\ p\ w$	$w \in U \cup B$	$\tau\ w$ `type Resource`
rdfs5t	$\tau_1\ v$ `subPropertyOf` w		$(\tau_1 \cup \tau_2)\ v$ `subPropertyOf` w
	$\tau_2\ w$ `subPropertyOf` u		$\tau_1\ v$ `subPropertyOf` u
rdfs6t	$\tau\ v$ `type Property`		$\tau\ v$ `subPropertyOf` v
rdfs7xt	$\tau_1\ p$ `subPropertyOf` q		$(\tau_1 \cup \tau_2)\ p$ `subPropertyOf` q
	$\tau_2\ v\ p\ w$	$q \in U \cup B$	$\tau_2\ v\ q\ w$
rdfs8t	$\tau\ v$ `type Class`		$\tau\ v$ `subClassOf Resource`
rdfs9t	$\tau_1\ v$ `subClassOf` w		$(\tau_1 \cup \tau_2)\ v$ `subClassOf` w
	$\tau_2\ u$ `type` v		$\tau_2\ u$ `type` w
rdfs10t	$\tau\ v$ `type Class`		$\tau\ v$ `subClassOf` v
rdfs11t	$\tau_1\ v$ `subClassOf` w		$(\tau_1 \cup \tau_2)\ w$ `subClassOf` u
	$\tau_2\ w$ `subClassOf` u		$\tau_1\ v$ `subClassOf` u
rdfs12t	$\tau\ v$ `type Cont.Memb.Prop.`		$\tau\ v$ `subPropertyOf member`
rdfs13t	$\tau\ v$ `type Datatype`		$\tau\ v$ `subClassOf Literal`

Table 7.1: $RDFS^\tau$ entailment rules based on the RDFS rules presented by H.J. ter Horst [ter Horst, 2005]

Note that we define the union of two intervals $i = [i_s, i_e], j = [j_s, j_e]$ as

$$i \cup j = [min(i_s, j_s), max(i_e, j_e)] \tag{7.7}$$

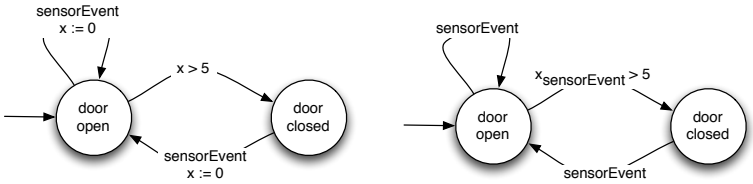
The union operation is applied whenever a RDFS rule is based on two different conditions. For instance, rule `rdfs5t` which entails the transitivity of a property's inheritance relations. If a subproperty has a temporal annotation that exceeds the superproperty's validity, we expand the superproperty's interval of validity with the union interval.

7.5 Dynamic Temporal Reasoning

In this section we are going to introduce Dynamic Temporal Concepts (DTC). DTC's have the characteristic that their entailment is dependent on the position of the timeline and the relation of entities to each other over time. Examples are duration-relations and patterns of occurrence in time. While static temporal reasoning can be achieved by evaluating rules and basic arithmetic (cf. Section 7.4), dynamic temporal reasoning require the "walking-through" the temporal dimension. Therefore, we will be employing the concept and theory of timed automata, more precisely the decidable sublcass of event clock automata [Alur et al., 1999] in the first sub-section. The following subsections first introduce event clock automata and then define the basic DTC building-blocks—duration, density, coverage, and repetition—which can serve as basic building blocks for DTCs. The last subsection discusses reasoning with DTCs.

7.5.1 Event Clock Automata

Timed automata introduce the notion of time variables that can be used as part of a condition. Consider the following simple example: An automatic door closes 5 seconds after the last person walked through it. This translates to the timed automata shown in Figure 7.2a. Every time a sensor-event occurs (i.e., a person passes through the door) the time variable x is reset to 0. If there is no sensor-event for more than 5 seconds then the door is closed, whereas the doors are immediately opened if a sensor-event occurs if the door was closed before. The disadvantage of these kind of automata is



(a) Timed automaton: An automatic door closes if there is no sensor input for 5 seconds.

(b) The equivalent event clock automaton for Figure 7.2a

Figure 7.2: Timed automata vs. event clock automata

their non-determinism. It is easy to construct temporal loops and dead-ends when variables are writable. The deterministic subclass of timed automata are the event clock automata (ECA). Determinism, however, comes at a costs: Variables are no longer allowed to be set during automata transitions. For every event there is one corresponding temporal variable that holds the amount of time units passed since the occurrence of the event. Furthermore, a deterministic form of a timed automaton must not contain ambiguous transition conditions. Hence, it cannot have a transition restriction of, e.g.,

$x > 2$ for one edge and $x < 4$ for another one as it would be unclear which edge to follow when $x = 3$.

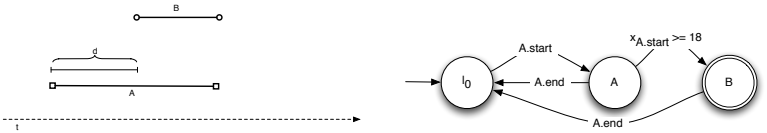
More formaly (and assuming that the reader to be familiar with the basic automata theory): An automata changes its state if a defined condition is met. A timed word $\overline{w}$ over an alphabet Σ is a (finite) sequence $(a_0, t_0)(a_1, t_1)...(a_n, t_n)$ with $a_i \in \Sigma$ with a non-decreasing order on t_i. A timed language is a set of timed words over Σ such that $\mathcal{L} \subseteq \Sigma$. A timed automata is a set of locations (or states) $l \in L$ and edges $e \in E$. Locations can be start locations ($L^0 \subseteq L$) and accepting locations ($L^f \subseteq L$). Each edge is a quadruple (l, l', a, φ) with the source location $l \in L$, the target location $l' \in L$, an input symbol $a \in \Sigma$ and a clock constraint φ (e.g. ≤ 3). The automaton changes state from l to l' if $\gamma_i^{\overline{w}} \models \varphi$ holds, where $\gamma_i^{\overline{w}}$ is the clock evaluation function for the timed word $\overline{w}$ at the time instant i.

In a next step we are going to bridge the gap between the graph definitions and the automata theory. Since every reasoning task happens on triple level we can map the notion of a triple $\langle s\ p\ o \rangle$ to the the automata-theoretic definition of a symbol $a \in \Sigma$. Since our temporal approach annotates triples with intervals $[i_1, i_2]$ which is not supported by timed automata, we rewrite a triple as two timed symbols: $a_{i_1}^s$ and $a_{i_2}^e$, where the first denotes the event when a triple became valid and the latter the moment when its validity ends. Consequently, an automaton can change states based on the start or end event of a triple. A graph G, as a finite set of triples, and in particular a temporal graph G_τ as a finite set of temporal triples relates to Σ^* – a finite set of timed words. The language that is accepted by a automaton is a set of timed words W with $W \subseteq S_\tau \subseteq G_\tau$ that participates in the entailment of G as $S_\tau \models_{\tau_{dynamic}} G_\tau$.

7.5.2 Basic Temporal Patterns

In this subsection we are going to describe a set of temporal patterns that can be directly translated to event clock automata in order compute valid entailments over G_τ.

Duration The first temporal concepts that we are going to introduce is the duration pattern. It simply matches whenever the validity of the described entity exceeds a given duration value. A simple example would be the equivalent class Adult consisting of the condition that the property `rdf:type Human` needs to be present for 18 or more years. Based on a triple `[1990, now] John rdf:type Human`, the DTC reasoner can entail the additional triple `[2008, now] John rdf:type Adult`. Figure 7.3a shows a visualization on a time line of a duration pattern. We can entail B if the base fact A is valid longer than d until the end of the validity of A. This pattern translates to a deterministic timed automata as showed in Figure 7.3b.



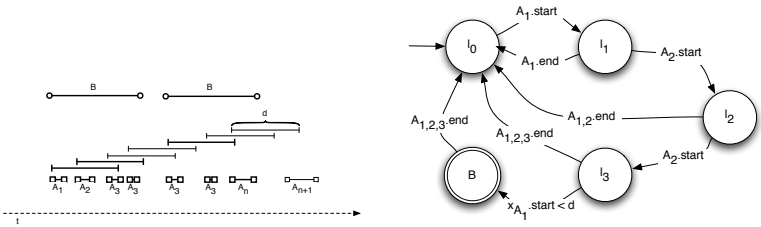
(a) The duration pattern matches entities of a certain age

(b) The pattern of Figure 7.3a translated to an event clock automaton

Figure 7.3: The duration pattern

Density The density pattern defines an event coverage of a certain time interval. Figure 7.4 shows an example. B is entailed as long as there is a period d in which the events A_n occurs at least x times. This pattern maps to the event clock automaton shown in Figure

7.4b. Note that this pattern has different interpretation possibilities. Since the patterns can be applied to interval annotated data an interval can be completely contained in period d, it can overlap d, or only the start or end points of the interval can be contained in d. A last possibility is that d is completely contained in at least x intervals. Figure 7.4 shows an example of an overlap interpretation. Consider the following example to elaborate on the different interpretations: A company is considered to be a `LargeCompany` if it has at least 1000 employees during a period of 3 months ($d = 3months$ has to be contained in at least 1000 worksFor relations). If more than 10 people start working for a company it is considered to be a `GrowingCompany` (at least 10 worksFor relations with a start point within $d = 1month$), and a company where more than 10 people quit in a month is considered to be a `ShrinkingCompany` (at least 10 worksFor relations with an end point within $d = 1month$).



(a) The density pattern matches whenever a certain amount of events occur in a defined period of time.

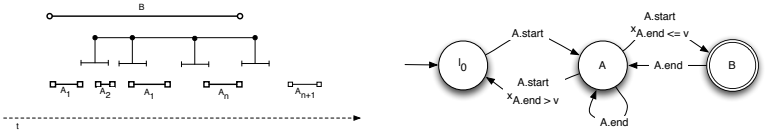
(b) The pattern of Figure 7.4a translated to an event clock automaton

Figure 7.4: The density pattern

The density pattern maps to a event clock automaton. Figure 7.4b shows the corresponding automaton for the example in Figure 7.4. Instead of counting the occurrences, we create as many locations (states) as there are defined by the pattern. If the number of events

equals x and the time difference between the first occurrence is $\leq d$, the automata matches the pattern.

Coverage The coverage pattern looks for the presence of a certain relation allowing defined periods of non-existence. Consider Figure 7.5. B is entailed if a relation A_n is present with a maximum absence thereof of at maximum v. An example for such a pattern could be the definition of a service level agreement. A server must be responsive all the times. It can be unavailable for maximum 15 minutes at a stretch. If it is longer unavailable than 15 minutes the SLA's are considered to be broken.



(a) The coverage pattern matches if a relation's validity is not absent for a longer than v.

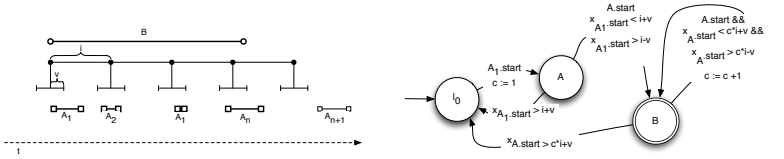
(b) The pattern of Figure 7.5a translated to an event clock automaton

Figure 7.5: The coverage pattern

Figure 7.5b shows the corresponding event clock automaton which matches the pattern described in Figure 7.5a.

Repetition A last temporal pattern is repetition. The pattern matches whenever an event occurs at a certain frequency (within a definable tolerance window). The pattern defines an interval i which is the pace of the frequency. As long as an event happens within the period of tolerance ($\pm v$) the pattern matches. Again, several interpretations are possible. Figure 7.6b shows an example with a `starts` interval interpretation. Consider the following example of a repe-

tition pattern that entails a new relation between two individuals. Two persons are considered to be `inContact` to each other if they send each other at least one email message a week (with a tolerance of ± 3 days). Note that Figure 7.6b shows the corresponding event clock automaton which is combined with an event count automaton [Chakraborty and Phan, 2006]. Since the count variable is only reset at the initial state, the automaton is still deterministic. The automaton matches the repetition pattern of Figure 7.6a. Next we are briefly going to investigate the satisfiability of a DTC followed by a vocabulary extension for persistent representation of the patterns.



(a) The repetition pattern matches if an event occurs at a certain frequency

(b) The pattern of Figure 7.6a translated to an event clock automaton

Figure 7.6: The repetition pattern

7.5.3 Satisfiability Checking

One of the major concerns in ontology creation and reasoning is the automated checking whether a concept is logically sound and does not contradict itself. In other words, the fact that there is a possible set of triples that entail the concept. In the world of automata theory this problem is described as the check whether an automata A defines a language $\mathcal{L}$ such that $\mathcal{L} \neq \{\emptyset\}$. To that end, the so-called region construction can be applied. This allows the automated checking for emptiness of the defined language of an event clock automa-

ton. This can be used in a reasoner for two purposes: firstly, it can serve as an optimization step to omit the checking for patterns that are unsatisfiable or, as a way of feedback to the user, a reasoner can issue a warning if such a pattern is encountered. For space considerations we are not further elaborating on this problem and refer the reader to [Alur et al., 1999].

7.6 Vocabulary Extension

In order to embed the above patterns in e.g. an ontology we are going to define a vocabulary extension for OWL in this section. As a base concept we extend OWL's Restriction class which we call `TemporalRestriction`. Whenever an equivalent class or property are defined using the new TemporalRestriction class, the defined pattern gets translated into an event clock automaton in order to entail new relations[1]. The class `TemporalRestriction` has four subclasses which relate to the four basic DTC patterns which we presented in Section 7.5. For the definition of durations we link to the OWL-Time ontology (denoted by the namespace prefix `time:`). For the temporally annotated data itself we are assuming a serialization format such as proposed in our preliminary publication [Tappolet and Bernstein, 2009].

[1]Note that we use our own namespace (dtc:) which relates to `http://www.ifi.uzh.ch/ddis/dtc/`, you can find the ontology online at `http://www.ifi.uzh.ch/ddis/dtc.html`

```
dtc:TemporalRestriction a       rdfs:Class ;
rdfs:subClassOf owl:Restriction .
dtc:CoverageRestriction  rdfs:subClassOf dtc:TemporalRestriction .
dtc:DurationRestriction  rdfs:subClassOf dtc:TemporalRestriction .
dtc:RepetitionRestriction rdfs:subClassOf dtc:TemporalRestriction .
dtc:DensityRestriction   rdfs:subClassOf dtc:TemporalRestriction .
dtc:AllenRelation       rdfs:subClassOf rdfs:Class .
dtc:finishes     a       dtc:AllenRelation .
dtc:finishes_inverse     a       dtc:AllenRelation .
dtc:overlaps     a       dtc:AllenRelation .
dtc:overlaps_inverse     a       dtc:AllenRelation .
dtc:starts       a       dtc:AllenRelation .
dtc:starts_inverse       a       dtc:AllenRelation .
dtc:during       a       dtc:AllenRelation .
dtc:during_inverse       a       dtc:AllenRelation .
dtc:meets        a       dtc:AllenRelation .
dtc:meets_inverse        a       dtc:AllenRelation .
dtc:before       a       dtc:AllenRelation .
dtc:after        a       dtc:AllenRelation .
dtc:equals       a       dtc:AllenRelation .
dtc:intervalSemantics    a       rdf:Property  ;
      rdfs:domain dtc:TemporalRestriction ;
      rdfs:range dtc:AllenRelation .
dtc:maxGap      a       rdf:Property ;
      rdfs:domain dtc:CoverageRestriction ;
      rdfs:range time:DurationDescription .
dtc:minDuration  a       rdf:Property  ;
      rdfs:domain dtc:DurationRestriction ;
      rdfs:range time:DurationDescription .
dtc:repetitionInterval  a       rdf:Property  ;
      rdfs:domain dtc:RepetitionRestriction ;
      rdfs:range time:DurationDescription .
dtc:repetitionTolerance a       rdf:Property ;
      rdfs:domain owl:RepetitionRestriction ;
      rdfs:range time:DurationDescription .
dtc:within      a       rdf:Property  ;
      rdfs:domain dtc:DensityRestriction ;
      rdfs:range time:DurationDescription .
```

When using the above vocabulary in combination with OWL syntax to describe equivalent classes or properties, a temporal rea-

soner can compile those patterns into event clock automata. Consider the following example which defines the equivalent class `Globetrotter` which is a person who changes its city of residence more than 5 times in 10 years (also used in the following evaluation section).

```
:Globetrotter
  a owl:Class ;
  owl:equivalentClass
    [ a dtc:DensityRestriction ;
      owl:onProperty :livesIn ;
      owl:hasMinValue 5^^xsd:Integer ;
      dtc:within
        [ a time:DurationDescription ;
            time:years 10 .]
      dtc:intervalSemantics dtc:overlaps ;
    ]
```

7.7 Evaluation

We implemented a prototype reasoner in order to entail the patterns that we described in Section 7.5. We used a straightforward implementation which still has potential for improvement. We did not implement the reasoner as a multi-threaded application although, if the temporal patterns are not related to each other, they could be entailed in parallel on multiple cores.

In order to evaluate this reasoner, we created an artificial temporal dataset where we randomly injected different temporal patterns (i.e., matches of temporal concepts). Specifically, the dataset covers 100 years (i.e., from 1911 until 2011) with a temporal resolution of days. Every day, new `Human`s start existing (i.e., are born). The number of new humans is randomly determined (with a gaussian dis-

tributed between 0 and 2). The age for every person is defined randomly (with a gaussian distribution between 0 and 120 with mean at 80 years). Humans regularly move to new cities. On average they live for 5 years in a City (randomly chosen). Finally, Humans communicate with each other. Every person has in average 1 communication relation to another person per day. Those relations can be sendsMail, calls, talksTo or visits.

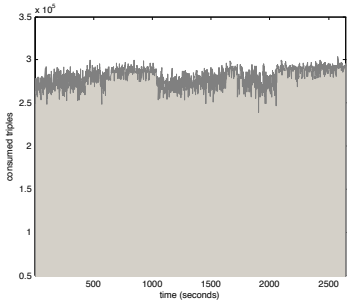
This procedure results in an average population of 29'200 Humans with the same number of communication relations and a number of moves to a new city (Note that the full population of 29'200 persons is gradually reached over 80 years). The procedure generates an average "stream size" of more than 20'000 triples per day which results in a dataset size of 742,96 million triples for the 100 years. In addition, we specifically injected solutions for the patterns we test the reasoner to match. This is to make sure that the reasoner is complete.

Since the temporal dimension allows a natural ordering of the triples, we sorted the dataset based on the start of the validity of triples. This is also the expected input format for the reasoner. The sorting effort was not counted as part of the reasoning process as this functionality can be efficiently supported by a storage system.

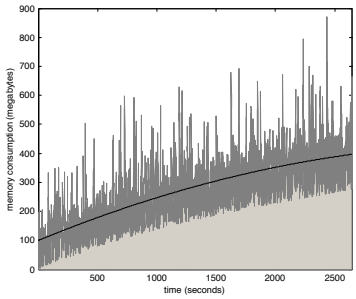
We defined temporal concepts based on the patterns described in Section 7.5. The patterns are: Nonagenarian (a Human with a duration (age) of more than 90 years), HomesickPerson (the duration in which a person did not leave its city for more than 3 months), Globetrotter (a person who changes its resident city more than 5 times in 10 years) and inContact (two persons are considered to be in contact to each other if they communicate, i.e. sendsMail, call, talkTo, visits on a weekly basis). We measured, the runtime for the entailment of every pattern and its memory consumption. Ad-

ditionally we compared the entailed set of triples with the expected solutions that we injected into the dataset. Figure **??** shows the results of the evaluation run over time.

We ran the evaluation on a machine with eight 3 GHz Intel Xeon quad core processors with 72 GB of RAM (note that we only assigned 8 GB of RAM to the Java Virtual Machine where the program was executed in) and a RAID 5 disk array. The reasoner performed equally over the whole dataset and reached a throughput of avg. 280'000 temporal triples per second (std. dev. = 10'381). Hence, we can conclude that reasoner's theoretical linear time complexity is met. We also measured the memory consumption over time. As a characteristic of our dataset, the population grows over time. Hence the overall reasoning effort grows the further the program advances (note that the population also grows after the year 1991 where the average life expectancy is first met, but the growth slows down at this point). This can be observed in Figure 7.7b where the memory consumption over time is plotted. One can see that toward the end (right end of the figure) the memory consumption's growth decreases after a period of almost linear increase. This is the expected behavior based on the structure of the artificial dataset. The program requires a maximum of 869 MB RAM which is a peak consumption. The average is 272 MB with a std. dev. of 120,23 MB. In order to visualize the memory consumption trend we also fitted a polynomial function to the data (2. order) which is plotted as a solid black line. In conclusion, both runtime and memory consumption remain, in our opinion, in a reasonable range.



(a) The number of triples consumed over time. The average triples per second is 280′000 with a std.dev. of 10′304.



(b) The memory consumption over time. The average consumption is 272 MB (std. dev. 120,23 MB). The solid black line is a 2. order polynomial function fitted to the data to show the trend.

Figure 7.7: Evaluation results

7.8 Conclusions, Limitations and Future Work

The Semantic Web is missing a temporal or at least a versioned perspective on its modeled domains. With temporal extensions, a knowledge engineer can add intervals of validity to triples in a knowledge base. In this paper we demonstrated how this validity information can be used to entail new facts and detect patterns in over time. We believe that temporal extensions are needed in a rapidly evolving environment such as the Linked Open Data movement. By bringing the temporal information to a symbolic level, standard reasoners can further process the information and entail even more knowledge. We presented two different kinds of temporal reasoning: static and dynamic reasoning. Static reasoning covers the entailment of validities along the existing entailment rules. Dynamic reasoning matches patterns of occurrences over time and allows the naming of these patterns using similar mechanisms as in

OWL. By mapping the detection of patterns to event clock automata (a subset of timed automata) we benefit from linear-time reasoning performance with a space complexity that is dependent on the defined patterns and not the number of triples.

This dependence is also a limitation of the approach because patterns could be defined that require the whole graph to be held in memory as background knowledge in order to match a pattern. As we showed in our evaluation with a realistic knowledge base, the space complexity is not a limiting factor on todays computers.

Further interesting challenges will be the complete integration of static and dynamic temporal reasoning into, e.g. an OWL reasoner. This will lead the way to a seamless integration of the temporal dimension into the Semantic Web.

Bibliography

[Abadi et al., 2007] Abadi, D. J., Marcus, A., Madden, S. R., and Hollenbach, K. (2007). Scalable Semantic Web Data Management Using Vertical Partitioning. In *33rd International Conference on Very Large Data Bases (VLDB)*.

[Allen, 1983] Allen, J. F. (1983). Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11).

[Alur et al., 1999] Alur, Y. R., Fix, Z. L., and Thomas, A. (1999). Event-clock automata: A determinizable class of timed automata. *Theoretical Computer Science*, 211(1-2):253–273.

[Anicic et al., 2011] Anicic, D., Fodor, P., Rudolph, S., and Stojanovic, N. (2011). Ep-sparql: a unified language for event processing and stream reasoning. In *Proceedings of the 20th international conference on World wide web*, pages 635–644. ACM.

[Artale and Franconi, 2005] Artale, A. and Franconi, E. (2005). Temporal description logics. In *Handbook of Time and Temporal Reasoning in Artificial Intelligence*. Elsevier.

[Barbieri et al., 2010] Barbieri, D., Braga, D., Ceri, S., and Valle, E. D. (2010). Incremental reasoning on streams and rich background

knowledge. *Proceedings of the 7th Extended Semantic Web Conference (ESWC)*.

[Bayer, 1997] Bayer, R. (1997). The universal B-tree for multidimensional indexing: General concepts. *Worldwide Computing and Its Applications*, pages 198–209.

[Bayer and Markl, 1998] Bayer, R. and Markl, V. (1998). The UB-tree: Performance of multidimensional range queries. Technical report.

[Bellman, 1961] Bellman, R. (1961). Dynamic programming. 1957. *Adaptive Control Processes, A Guided Tour*.

[Berners-Lee and Hendler, 2001] Berners-Lee, T. and Hendler, J. (2001). Scientific publishing on the semantic web. *Nature*, 410:1023–1024.

[Bernstein et al., 2007a] Bernstein, A., Ekanayake, J., and Pinzger, M. (2007a). Improving Defect Prediction Using Temporal Features and Non-linear Models. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE)*.

[Bernstein et al., 2007b] Bernstein, A., Kiefer, C., and Stocker, M. (2007b). OptARQ: A SPARQL Optimization Approach based on Triple Pattern Selectivity Estimation. Technical report, Department of Informatics, University of Zurich.

[Bizer et al., 2005] Bizer, C., Cyganiak, R., and Watkins, R. (2005). NG4J - Named Graphs API for Jena (Poster). In *Proc. of the 2nd European Semantic Web Conference*.

[Boehm, 1981] Boehm, B. W. (1981). *Software Engineering Economics*. Prentice Hall.

[Bossche et al., 2007] Bossche, M., Ross, P., MacLarty, I., Van Nuffelen, B., and Pelov, N. (2007). Ontology driven software engineering for real life applications. In *Proc. 3rd Intl. Workshop on Semantic Web Enabled Software Engineering*. Citeseer.

[Campbell et al.,] Campbell, C. E., Eisenberg, A., and Melton, J. Xml schema. Technical report, W3C Recommendation.

[Carroll et al., 2005] Carroll, J. J., Bizer, C., Hayes, P., and Stickler, P. (2005). Named graphs. *Journal of Web Semantics*, 3(3).

[Carroll et al., 2004] Carroll, J. J., Dickinson, I., Dollin, C., Seaborne, A., Wilkinson, K., and Reynolds, D. (2004). Jena: Implementing the semantic web recommendations.

[Chakraborty and Phan, 2006] Chakraborty, S. and Phan, L. (2006). Event count automata: A state-based model for stream processing systems. *Proceedings of the 26th IEEE International Real-Time Systems Symposium*.

[Chen et al., 2007] Chen, N., Wang, N., and Shi, B. (2007). A new algorithm for encoding and decoding the hilbert order. *Software: Practice and Experience*, 37(8):897–908.

[Chen et al., 1990] Chen, Y.-F., Nishimoto, M., and Ramamoorthy, C. (1990). The C Information Abstraction System. *IEEE Transactions on Software Engineering*, 16(3):325–334.

[Chenyang et al., 2008] Chenyang, L., Hong, Z., and Nengchao, W. (2008). Fast n-dimensional hilbert mapping algorithm. *Proceedings of the International Conference on Computational Sciences and Its Applications*, pages 507–513.

[Demeyer et al., 1999] Demeyer, S., Tichelaar, S., and Steyaert, P. (1999). FAMIX 2.0—the FAMOOS inf. exchange model. Technical report, University of Berne, Switzerland.

[Devanbu, 1999] Devanbu, P. (1999). GENOA - A Customizable, Front-end Retargetable Source Code Analysis Framework. *ACM Transactions on Software Engineering and Methodology*, 8:177–212.

[Devanbu et al., 1991] Devanbu, P., Brachman, R., and Selfridge, P. G. (1991). LaSSIE: a Knowledge-based Software Information System. *Communications of the ACM*, 34:34–49.

[Dietrich and Elgar, 2005] Dietrich, J. and Elgar, C. (2005). A formal description of design patterns using OWL. In *Proceedings of the Australian Software Engineering Conference*, Brisbane, Australia.

[Džeroski, 2003] Džeroski, S. (2003). Multi-relational data mining: An introduction. *ACM SIGKDD Explorations Newsletter*, 5(1):1–16.

[Ebert et al., 2002] Ebert, J., Kullbach, B., Riediger, V., and Winter, A. (2002). GUPRO: Generic Understanding of Programs – an Overview. *Electronic Notes in Theoretical Computer Science*.

[Elmasri et al., 1990] Elmasri, R., Wuu, G., and Kim, Y. (1990). The time index: An access structure for temporal data. In *Proc. of the 16th Int'l Conference on Very Large Data Bases*.

[Emerson, 1990] Emerson, E. A. (1990). Temporal and modal logic. In *Handbook of Theoretical Computer Science*. Elsevier.

[Feijs et al., 1998] Feijs, L., Krikhaar, R., and Van Ommering, R. (1998). A relational approach to support software architecture analysis. *Software-Practice and Experience*, 28(4):371–400.

[Fenton and Neil, 1999] Fenton, N. and Neil, M. (1999). A Critique of Software Defect Prediction Models. *IEEE Transactions On Software Engineering*, 25(3).

[Fenton, 1991] Fenton, N. E. (1991). *Software Metrics: A Rigorous Approach*. Chapman & Hall, Ltd.

[Fischer et al., 2003] Fischer, M., Pinzger, M., and Gall, H. (2003). Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 23–32, Amsterdam, Netherlands.

[Fowler, 1999] Fowler, M. (1999). *Refactoring*. Addison-Wesley.

[Gaede and Günther, 1998] Gaede, V. and Günther, O. (1998). Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231.

[Gamma et al., 1995] Gamma, E., Helm, R., and Johnson, R. E. (1995). *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman.

[Goldschlager, 1981] Goldschlager, L. (1981). Short algorithms for space-filling curves. *Software: Practice and Experience*, 11(1):99–99.

[Gröner et al., 2008] Gröner, G., Staab, S., and Winter, A. (2008). Graph Technology and Semantic Web in Reverse Engineering – A Comparison. In *Proceedings of ICPC 2008 Workshop: Semantic Technologies in System Maintenance*.

[Gutierrez et al., 2005] Gutierrez, C., Hurtado, C., and Vaisman, A. (2005). Temporal RDF. In *Proc. of the 2nd European Semantic Web Conference*.

[Gutierrez et al., 2007] Gutierrez, C., Hurtado, C. A., and Vaisman, A. (2007). Introducing time into RDF. *IEEE Transactions on Knowledge and Data Engineering*, 19(2).

[Guttman, 1984] Guttman, A. (1984). R-trees: a dynamic index structure for spatial searching. *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57.

[Happel et al., 2006] Happel, H.-J., Korthaus, A., Seedorf, S., and Tomczyk, P. (2006). KOntoR: An Ontology-enabled Approach to Software Reuse. In *Proceedings of the 18th Internatinal Conference on Software Engineering and Knowledge Engineering (SEKE)*, San Francisco, CA.

[Heath and Bizer, 2011] Heath, T. and Bizer, C. (2011). *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool, 1st edition.

[Hilbert, 1891] Hilbert, D. (1891). Ueber die stetige Abbildung einer Line auf ein Flächenstück. *Mathematische Annalen*.

[Hobbs and Pan, 2005] Hobbs, J. R. and Pan, F. (2005). Time ontology in OWL. Technical report, W3C Working Draft.

[Hoffart et al., 2010] Hoffart, J., Suchanek, F. M., Berberich, K., and Weikum, G. (2010). Yago2: a spatially and temporally enhanced knowledge base from wikipedia. Research Report MPI-I-2010-5-007, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany.

[Holt, 1998] Holt, R. C. (1998). Structural manipulations of software architecture using tarski relational algebra. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, page 210. IEEE Computer Society.

[Holt et al., 2000] Holt, R. C., Winter, A., and Schürr, A. (2000). GXL: Toward a Standard Exchange Format. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE)*.

[Hyland-Wood et al., 2006] Hyland-Wood, D., Carrington, D., and Kapplan, S. (2006). Toward a Software Maintenance Methodology using Semantic Web Techniques. In *Proceedings of the 2nd ICSM International Workshop on Software Evolvability (SE)*, pages 23–30.

[Hyvönen, 2007] Hyvönen, E. (2007). Modeling and reasoning about changes in ontology time series. *Integrated Series in Information Systems*, 14 II:319–338.

[Jagadish, 1990] Jagadish, H. V. (1990). Linear clustering of objects with multiple attributes. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, SIGMOD '90, pages 332–342, New York, NY, USA. ACM.

[Kamel and Faloutsos, 1994] Kamel, I. and Faloutsos, C. (1994). Hilbert R-tree: An improved R-tree using fractals. *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 500–509.

[Kiefer et al., 2008] Kiefer, C., Bernstein, A., and Locher, A. (2008). Adding Data Mining Support to SPARQL via Statistical Relational Learning Methods. In *Proceedings of the 5th European Semantic Web Conference (ESWC)*. Springer.

[Kiefer et al., 2007a] Kiefer, C., Bernstein, A., and Stocker, M. (2007a). The Fundamentals of iSPARQL — A Virtual Triple Approach For Similarity-Based Semantic Web Tasks. In *Proceedings of the 6th International Semantic Web Conference (ISWC)*.

[Kiefer et al., 2007b] Kiefer, C., Bernstein, A., and Tappolet, J. (2007b). Analyzing Software with iSPARQL. In *Proceedings of the 3rd International Workshop on Semantic Web Enabled Software Engineering (SWESE)*. Springer.

[Kiefer et al., 2007c] Kiefer, C., Bernstein, A., and Tappolet, J. (2007c). Mining Software Repositories with iSPARQL and a Software Evolution Ontology. In *Proceedings of the 4th ICSE International Workshop on Mining Software Repositories (MSR)*.

[Kim et al., 2008] Kim, S.-K., Song, M.-Y., Kim, C., and Jang, H. C. (2008). Temporal ontology language for representing and reasoning interval-based temporal knowledge. In *Proc. of the 3rd Asian Semantic Web Conference*.

[Klyne and Carroll, 2004a] Klyne, G. and Carroll, J. J. (2004a). Resource description framework (RDF): Concepts and abstract syntax. World Wide Web Consortium, Recommendation REC-rdf-concepts-20040210.

[Klyne and Carroll, 2004b] Klyne, G. and Carroll, J. J. (2004b). Resource description framework (RDF): Concepts and abstract syntax. Technical report, W3C Recommendation.

[Koubarakis and Kyzirakos, 2010] Koubarakis, M. and Kyzirakos, K. (2010). Modeling and querying metadata in the Semantic Sensor Web: the model stRDF and the query language stSPARQL. *Proceedings of the 7th Extended Semantic Web Conference (ESWC)*.

[Lanza and Marinescu, 2006] Lanza, M. and Marinescu, R. (2006). *Object-Oriented Metrics in Practice*. Springer.

[Lawder, 2001] Lawder, J. K. (2001). Querying multi-dimensional data indexed using the Hilbert space-filling curve. *SIGMOD Record*, 30:2001.

[Lethbridge et al., 2003] Lethbridge, T. C., Singer, J., and Forward, A. (2003). How Software Engineers Use Documentation: The State of the Practice. *IEEE Software*, 20(6):35–39.

[Lethbridge et al., 2004] Lethbridge, T. C., Tichelaar, S., and Ploedereder, E. (2004). The Dagstuhl Middle Metamodel: A Schema For Reverse Engineering. In *Proceedings of the International Workshop on Meta-Models and Schemas for Reverse Engineering*.

[Levenshtein, 1966] Levenshtein, V. I. (1966). Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707–710.

[Li and Zhang, 2011] Li, Y.-F. and Zhang, H. (2011). Integrating software engineering data using semantic web technologies. In *Proceeding of the 8th working conference on Mining software repositories*, MSR '11, pages 211–214, New York, NY, USA. ACM.

[Lopes et al., 2010] Lopes, N., Polleres, A., Straccia, U., and Zimmermann, A. (2010). Anql: Sparqling up annotated rdfs. *The Semantic Web–ISWC 2010*, pages 518–533.

[Mäntylä et al., 2003] Mäntylä, M., Vanhanen, J., and Lassenius, C. (2003). A Taxonomy and an Initial Empirical Study of Bad Smells in Code. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 381–384.

[Milea et al., 2008] Milea, V., Frasincar, F., and Kaymak, U. (2008). Knowledge engineering in a temporal semantic web context. In *The Eighth Int'l Conference on Web Engineering*.

[Motik, 2010] Motik, B. (2010). Representing and querying validity time in RDF and OWL: A logic-based approach. *Proceedings of the 9th Internation Semantic Web Conference (ISWC)*.

[Müller and Klashinsky, 1988] Müller, H. A. and Klashinsky, K. (1988). RIGI – A System for Programming-in-the-large. In *Proceedings of the 10th international Conference on Software Engineering (ICSE)*.

[Nascimento and Dunham, 1999] Nascimento, M. and Dunham, M. (1999). Indexing valid time databases via B+-trees. *IEEE Transactions on Knowledge and Data Engineering*, 11(6):929–947.

[Neville et al., 2003a] Neville, J., Jensen, D., Friedland, L., and Hay, M. (2003a). Learning Relational Probability Trees. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 625–630.

[Neville et al., 2003b] Neville, J., Jensen, D., and Gallagher, B. (2003b). Simple Estimators for Relational Bayesian Classifiers. In *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM)*, pages 609–612.

[O'Connor et al., 2007] O'Connor, M., Shankar, R., Parrish, D., and Das, A. (2007). Knowledge-level querying of temporal patterns in clinical research systems. In *Proc. of the 12th World Congress on Health (Medical) Informatics*.

[OMG, 1998] OMG (1998). XML Metadata Interchange (XMI). Technical report, Object Management Group.

[OMG, 2008] OMG (2008). Query / View / Transformation (QVT). Technical report, Object Management Group.

[Pan, 1995] Pan, D. (1995). A tutorial on mpeg/audio compression. *IEEE MultiMedia*, 2(2).

[Perry, 2008] Perry, M. (2008). *A framework to support spatial, temporal and thematic analytics over semantic web data*. PhD thesis, Wright State University.

[Provost and Fawcett, 2001] Provost, F. J. and Fawcett, T. (2001). Robust Classification for Imprecise Environments. *Machine Learning*, 42(3):203–231.

[Prud'hommeaux and Seaborne, 2008] Prud'hommeaux, E. and Seaborne, A. (2008). SPARQL Query Language for RDF. Technical report, W3C.

[Prud'hommeaux and Seaborne, 2008] Prud'hommeaux, E. and Seaborne, A. (2008). SPARQL query language for RDF. Technical report, W3C Recommendation.

[Pugliese et al., 2008] Pugliese, A., Udrea, O., and Subrahmanian, V. S. (2008). Scaling RDF with time. In *Proc. of the 17th Int'l World Wide Web Conference*.

[Raimondi et al., 2008] Raimondi, F., Skene, J., and Emmerich, W. (2008). Efficient online monitoring of web-service SLAs. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering (FSE)*.

[Randell et al., 1992] Randell, D., Cui, Z., and Cohn, A. (1992). A spatial logic based on regions and connection. *Knowledge Representation*, 92:165–176.

[Royce, 1970] Royce, W. (1970). Managing the development of large software systems. In *proceedings of IEEE WESCON*, volume 26. Los Angeles.

[Sager et al., 2006] Sager, T., Bernstein, A., Pinzger, M., and Kiefer, C. (2006). Detecting similar java classes using tree algorithms. In *Proceedings of the 3rd ICSE International Workshop on Mining Software Repositories (MSR)*, pages 65–71.

[Schürr et al., 1999] Schürr, A., Winter, A. J., and Zündorf, A. (1999). The PROGRES Approach: Language and Environment. *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 2: Applications, Languages, and Tools*, pages 487–550.

[Shatnawi and Li, 2006] Shatnawi, R. and Li, W. (2006). A Investigation of Bad Smells in Object-Oriented Design Code. In *Proceedings of the 3rd International Conference on Information Technology: New Generations*, pages 161–165.

[Shen et al., 1994] Shen, H., Ooi, B., and Lu, H. (1994). The TP-Index: A dynamic and efficient indexing mechanism for temporal databases. *International Conference on Data Engineering*, pages 274–274.

[Snodgrass, 1995] Snodgrass, R. T., editor (1995). *The TSQL2 Temporal Query Language*. Kluwer.

[Song and Roussopoulos, 2002] Song, Z. and Roussopoulos, N. (2002). Using Hilbert curve in image storing and retrieving. *Information Systems*, 27(8):523–536.

[Stantic et al., 2010] Stantic, B., Topor, R., Terry, J., and Sattar, A. (2010). Advanced indexing technique for temporal data. *Computer Science and Information Systems*, 7(4):679–703.

[Stocker et al., 2008] Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., and Reynolds, D. (2008). SPARQL Basic Graph Pattern Opti-

mization Using Selectivity Estimation. In *Proceedings of the 17th International World Wide Web Conference (WWW)*.

[Tao et al., 2010] Tao, C., Solbrig, H., Sharma, D., Wei, W., Savova, G. K., and Chute, C. G. (2010). Time-oriented question answering from clinical narratives using Semantic-Web techniques. *Proceedings of the 9th International Semantic Web Conference*.

[Tappolet, 2008] Tappolet, J. (2008). Semantics-aware Software Project Repositories. In *ESWC 2008 Ph.D. Symposium*, pages 78–82. Best Presentation.

[Tappolet and Bernstein, 2009] Tappolet, J. and Bernstein, A. (2009). Applied temporal RDF: Efficient temporal querying of RDF data with SPARQL. *Proceedings of the 6th European Semantic Web Conference*.

[ter Horst, 2005] ter Horst, H. J. (2005). Completeness, decidability and complexity of entailment for RDF schema and a semantic extension involving the OWL vocabulary. *Journal of Web Semantics*, 3(2-3):79–115.

[Tropf and Herzog, 1981] Tropf, H. and Herzog, H. (1981). Multidimensional range search in dynamically balanced trees. *Angewandte Informatik*, pages 71–77.

[Valiente, 2002] Valiente, G. (2002). *Algorithms on Trees and Graphs*. Springer.

[Völkel et al., 2005] Völkel, M., Enguix, C. F., Kruk, S. R., Zhdanova, A. V., Stevens, R., and Sure, Y. (2005). Semversion - versioning RDF and ontologies. Technical report, Institute AIFB, University of Karlsruhe.

[Weiss et al., 2008] Weiss, C., Karras, P., and Bernstein, A. (2008). Hexastore: Sextuple Indexing for Semantic Web Data Management. In *Proc. of the 34th Intl Conf. on Very Large Data Bases (VLDB)*.

[Welty and Fikes, 2006] Welty, C. A. and Fikes, R. (2006). A reusable ontology for fluents in OWL. In *Proc. of the 4th Int'l Conference On Formal Ontology In Information Systems*.

[Witte et al., 2007] Witte, R., Zhang, Y., and Rilling, J. (2007). Empowering Software Maintainers with Semantic Web Technologies. In *In Proceedings of the 4th European Semantic Web Conference (ESWC)*.

[Witten and Frank, 2005] Witten, I. H. and Frank, E. (2005). *Data Mining : Practical Machine Learning Tools and Techniques*. Elsevier, Morgan Kaufman, 2. ed. edition.

[Zhang, 2007] Zhang, Y. (2007). *An Ontology-based Program Comprehension Model*. PhD thesis, Concordia University Montreal, Quebec, Canada.

List of Figures

List of Tables

List of Listings

Curriculum Vitae

Personal Information

Name	Lucien Jonas Tappolet
Date of birth	14.05.1982
Place of origin	Zurich, ZH
Citizenship	Swiss

Education

2011	Doctor of Science UZH
2007 – 2011	Doctoral Student, Dynamic and Distributed Information Systems, University of Zurich
2007	Master of Science UZH
2002 – 2007	Studies in Informatics and Business Administration (dipl. inform.), University of Zurich
2002	Matura Typus N
1997 – 2002	Maturitätsschule (high school), Kantonsschule Schaffhausen